UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung eines virtuellen Billardspiels

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Andreas Herschbach

Betreuer: Dipl.-Inform. Oliver Abert
Institut für Computervisualistik, AG Computergrafik

Koblenz, im Mai 2007

Erklärung

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Der Traum vieler Computervisualisten ist sicherlich der spätere Einstieg in die Spielebranche. Allerdings wird sich dieser Traum nicht für jeden erfüllen können, da nur wenige das Glück haben, in einer Spielefirma beschäftigt zu werden. Daher ist es wichtig, schon während des Studiums entsprechende Erfahrungen auf dem Gebiet der Spieleprogrammierung zu sammeln, was bei der Wahl des Themas dieser Studienarbeit auch eine große Rolle spielte.

Die Wahl fiel dabei auf ein Billardspiel. Der Entwicklungsaufwand hält sich hier in noch überschaubaren Grenzen, vereint aber dennoch alle wichtigen Dinge, die ein Spiel benötigt. Dazu gehört das Modellieren der Spielumgebung sowie das anschließende Laden derselben in das Programm. Auch eine passende Texturierung ist wünschenswert, um einen möglichst realistischen Eindruck zu erzeugen. Außerdem müssen alle Objekte im Spiel miteinander interagieren können. Die Kugeln können untereinander oder mit den Banden kollidieren, können in die Taschen fallen und angespielt werden. Für diese Interaktionsmöglichkeiten sowie die Berechnung der aktuellen Kugelgeschwindigkeiten wird eine Physikengine benötigt, die das Verhalten aller Objekte möglichst realistisch im Spiel nachbildet. Allerdings ist selbst die beste Physikengine nutzlos, wenn die Steuerung nicht möglichst intuitiv gelöst ist. Das Gefühl, den Queue wirklich in der Hand zu haben, ist durch nichts zu ersetzen. Außerdem sollen Soundeffekte einen weiteren Teil zur Atmosphäre des Spiels beitragen, und auch die Spielregeln vom Computer geprüft werden.

Diese Ausarbeitung gibt einen Einblick in die Entwicklung dieses Spiels, von den nötigen Grundlagen über die Modellierung und Texturierung bis hin zur Implementierung des Spiels und den Ergebnissen.

2 Grundlagen

Billard ist ein Spiel, das sehr stark von der Physik abhängig ist. Deshalb sollte die Physik ein keinem Billardspiel vernachlässigt werden. Die für diese Studienarbeit wichtigen Grundlagen aus dem Billardsport werden in diesem Kapitel erläutert.

2.1 Spielarten

Die klassischen Spielarten beim Billard sind *8-Ball* und *9-Ball Pool*. Neben diesen gibt es noch eine Reihe weiterer Variationen, wie beispielsweise *Snooker*, *Carambolage* oder *14.1 endlos*. Diese Arbeit konzentriert sich auf die beiden populärsten Spielarten, dem *8-Ball* und *9-Ball Pool*.

Beim *8-Ball Pool* befinden sich, neben der weißen Kugel, 15 weitere Kugeln auf dem Tisch. Dabei werden, neben der weißen Kugel, drei Arten unterschieden, volle Kugeln, halbe Kugeln und die Acht. Zu Beginn des Spiels

werden die Kugeln in einem Dreieck aufgebaut, sodass die Acht in der Mitte liegt. Hinten links und rechts müssen sich jeweils eine halbe und eine volle Kugel befinden, die Anordnung der restlichen Kugeln ist egal. Ein Spieler spielt auf die vollen Kugeln, einer auf die halben. Ziel ist es, alle Kugeln der eigenen Farbe und schließlich die Acht zu versenken. Die Reihenfolge der Kugeln der eigenen Farbe ist dabei egal. Allerdings muss bei jedem Stoß zuerst eine eigene Kugel berührt werden. Wird eine eigene Kugel eingelocht, ist der Spieler weiterhin an der Reihe. Wird keine Kugel eingelocht, muss die angespielte Kugel mindestens eine Bande berühren. Wird eine falsche Kugel zuerst oder keine Bande berührt oder die weiße Kugel versenkt, liegt ein Foul vor. In diesem Fall ist der nächste Spieler an der Reihe, der die weiße Kugel nun beliebig auf dem Tisch platzieren darf. Sind alle eigenen Kugeln versenkt, muss die schwarze Kugel in eine beliebige, aber angesagte Tasche gespielt werden. Wird hierbei die schwarze Kugel in eine falsche Tasche gespielt, ist das Spiel verloren.

Beim 9-Ball Pool befinden sich zu Spielbeginn die Kugeln mit den Nummern 1 bis 9, neben der weißen, auf dem Tisch. Zu Beginn werden die Kugeln diamantförmig aufgebaut, die Neun liegt dabei in der Mitte, die Eins vorne. Ziel des Spiels ist es, die Neun mit einem korrekten Stoß zu versenken. Beide Spieler spielen dabei auf alle Kugeln, einzige Beschränkung ist, dass die niedrigste auf dem Tisch befindliche Kugel zuerst angespielt werden muss. Es liegt ein Foul vor, wenn eine Kugel des Gegners zuerst oder keine Bande berührt wurde, oder die weiße Kugel versenkt wird. Direkt nach dem Eröffnungsstoß kann der Spieler, wenn er die niedrigste Kugel nicht direkt anspielen kann, *Push-Out* spielen. Dabei muss die weiße Kugel keine andere und keine Bande berühren, der Spieler kann die weiße also in jede beliebige Richtung spielen. Der gegnerische Spieler entscheidet dann, ob er die Position übernimmt, oder dem Spieler, der *Push-Out* gespielt hat, den Tisch überlässt. Fällt die Neun, auch unbeabsichtigt, ist das Spiel gewonnen.

2.2 Kollision und Kraftübertragung

Beim Billard gibt es nur eine Form der Kraftübertragung, den elastischen Stoß. In der Theorie kann sogar von einem ideal elastischen Stoß gesprochen werden, bei dem bei einer Kollision zweier Kugeln keine Bewegungsenergie verloren geht. Es gilt der Energieerhaltungssatz, nach dem die Summe der Bewegungsenergie vor und nach dem Stoß identisch sein muss (Quelle: [11]). Ist m die Masse eines Objekts, v die Geschwindigkeit vor und v' die Geschwindigkeit nach der Kollisions, so ist die Energie W vor dem Stoß gegeben durch Formel 1 und die Energie nach dem Stoß durch Formel 2. Nach dem Energieerhaltungssatz gilt folglich Formel 3.

$$W = \frac{1}{2} \cdot m_1 \cdot v_1^2 + \frac{1}{2} \cdot m_2 \cdot v_2^2 \tag{1}$$

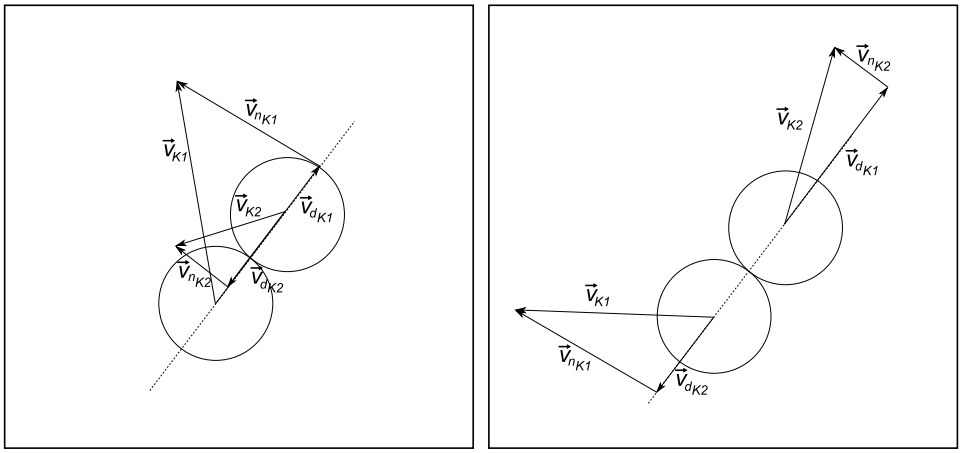

Abbildung 1: Die Bewegungsenergie der beiden Kugeln wird nach der Kollision in zwei Komponenten aufgeteilt, aus denen sich die neue Bewegungsenergie errechnen lässt.

$$W' = \frac{1}{2} \cdot m_1 \cdot v_1'^2 + \frac{1}{2} \cdot m_2 \cdot v_2'^2 \tag{2}$$

$$\frac{1}{2} \cdot m_1 \cdot v_1^2 + \frac{1}{2} \cdot m_2 \cdot v_2^2 = \frac{1}{2} \cdot m_1 \cdot v_1'^2 + \frac{1}{2} \cdot m_2 \cdot v_2'^2 \tag{3}$$

Im eindimensionalen Raum ist die Berechnung der neuen Energien noch recht einfach, im mehrdimensionalen Raum müssen die neuen Energien entsprechend aufgeteilt werden, um auch die neue Laufrichtung zu repräsentieren. Die Bewegungsenergie wird nach der Kollision aufgeteilt in eine Kraft parallel zur Verbindungslinie der Kugelmittelpunkt, $\vec{v}_d$, und eine Kraft senkrecht dazu, $\vec{v}_n$, wie in Abbildung 1 zu sehen. Nun wird $\vec{v}_d$ der beiden Kugeln vertauscht und durch Addition von $\vec{v}_n$ die neue Bewegungsenergie der Kugeln berechnet. Die genaue Formel hierzu wird in Kapitel 4.3.3 erläutert.

2.3 Effet

Mit Effet wird das dezentrale Anspielen der weißen Kugel bezeichnet, um die spätere Laufbahn dieser Kugel zu beeinflussen. Bei einem zentralen Anstoß gleitet die Kugel zunächst über die Oberfläche des Tischs, beginnt nach kurzer Zeit jedoch durch die Reibung mit der Spielfläche zu rollen. Bei einem Effetstoß wird die Kugel zusätzlich zum eigentlichen Impuls in Laufrichtung in eine Rotation um die eigene Achse versetzt. Kollidiert die Kugel mit einem anderen Objekt, wirkt sich der Effet auf die Laufrichtung der Kugel nach der Kollision aus.

Wird die Kugel oberhalb der Mitte angespielt, folgt sie der anderen Kugel nach der Kollision, daher wird dieser Stoß auch als *Nachläufer* bezeichnet. Wird die Kugel unterhalb der Mitte angespielt, rollt sie ein Stück zum Spieler zurück, daher wird dieser Stoß auch als *Rückläufer* bezeichnet. In beiden

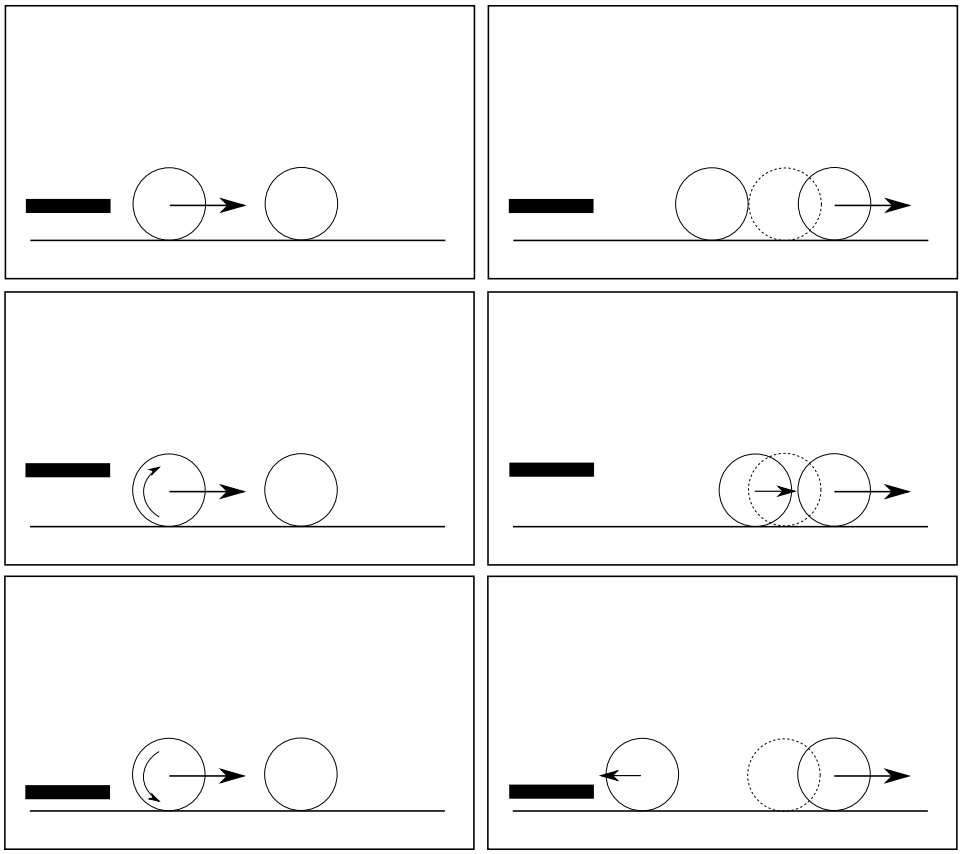

Abbildung 2: Beeinflussung der Laufrichtungen durch Effet. Oben wird die Kugel mittig angespielt, bei einer Kollision stoppt sie. In der Mitte wird die Kugel oben angespielt, durch den Effet rollt sie der anderen Kugel bei einer Kollision nach. Unten wird die Kugel unten angespielt, durch den Effet rollt sie nach der Kollision zurück.

Fällen stoppt die Kugel kurz, nachdem sie eine andere Kugel getroffen hat. Durch ihre Drehung sowie die Reibung auf dem Tisch beginnt sie dann wieder, sich in Rotationsrichtung zu bewegen. Abbildung 2 zeigt diese Effekte.

Der Effet beim seitlichen Anspielen, genannt *English*, wirkt sich nur bei einer Kollision mit einer Bande auf die Laufbahn aus. Bei geradem Auftreffen auf die Bande läuft die Kugel nach der Kollision in die Richtung, in der sie angespielt wurde. Bei einem schrägen Auftreffen ändert sich der Ausfallswinkel entsprechend dem Effet. Abbildung 3 zeigt die Veränderungen der Kugellaufbahn bei seitlichem Anspielen.

Wichtig bei diesen Effetstößen ist, den Stoß durchzuziehen, das heißt den Queue nach der Berührung des Spielballs noch ein Stück weiterzuführen. Bei einem kurzen, abrupten Stoß ist der Effet nur sehr gering und aller Wahrscheinlichkeit nach bis zum Erreichen des Ziels in normale Bewegungsenergie

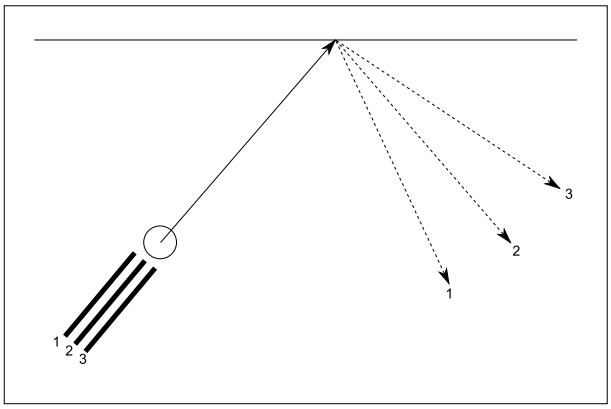

Abbildung 3: Beeinflussung der Laufrichtungen durch Effet. Die gestrichelten Linien zeigen die Laufbahn der Kugel nach der Kollision mit der Bande, wenn sie in Position 1, 2 und 3 angespielt wurde.

übergegangen.

Eine extreme Form des Effets ist ein sogenannter *Massestoß*. Hierbei wird die Kugel von schräg oben angespielt, um der Kugel einen besonders starken Effet zu versetzen. Dieser Effet ist so stark, dass die Kugel auch ohne Kollision mit einem anderen Objekt allein durch die Reibung mit dem Spielfeld ihre Laufbahn entsprechend ihrer Eigenrotation verändert. So ist es möglich, die Kugel Kurven laufen zu lassen.

3 Modellierung und Texturierung

Um das Spiel auch optisch ansprechend zu gestalten, ist eine möglichst realistische Gestaltung der Spielumgebung äußerst wichtig. Da dies allerdings einen nicht unerheblichen Aufwand bedeutet, ist es im Spiel nicht möglich, zwischen mehreren Tischgrößen zu wählen, es steht lediglich ein Tisch in Turniergröße bereit.

Da schon grundlegende Kenntnisse in der Modellierungs- und Animationssoftware *Maya 6.0* vorhanden waren, wurde dieses Tool für die Erstellung der Modelle benutzt. Die Texturen entstanden mit *Adobe Photoshop 7.0* und *Wood Workshop* und wurden anschließend in *Maya* eingebunden.

3.1 Tisch

Der Tisch wurde nach aktuell gültigen Turniernormen möglichst maßstabsgetreu nachgebaut. Der Einfachheit halber entspricht eine Längeneinheit einem Zentimeter.

Grundgerüst des Tischs bildet ein Quader der Größe 285x40x167, der um -16.4 in Y-Richtung verschoben wurde. So wurde sichergestellt, dass die

Spielfläche später in der XZ-Ebene liegt. Dieser Quader wurde an der Unterseite verkleinert, um eine leicht schräge Form zu erhalten. Die Kanten wurden abschließend mit einem *Bevel*[1] abgerundet. Anschließend wurde die Spielfläche der Größe 258x3.6x140 über eine *Boolean-Operation*[2] mit einem weiteren Quader herausgeschnitten. Auf dieselbe Art wurden die Taschen in den Tisch geschnitten. Dafür wurden Zylinder mit einem Durchmesser von 11.2 und einer Höhe von 23.6 erstellt und an die richtigen Positionen verschoben. Da die Taschen eines Billardtisches nicht ganz rund sind, wurden jeweils zwei Zylinder hintereinander gelegt, um eine längliche Form zu erzeugen. Die Grundform des Tisches ist somit fertig.

Die Banden bestehen ebenfalls aus Quadern, die mittels Boolen-Operations zurechtgeschnitten wurden. Die Quader der Seitenbanden haben die Maße 115.76x3.6x6.5, die Quader am Kopf und Fußende die Maße 124.5x3.6x6.5. Die Innenkanten der Bande besitzen einen Winkel von 60° zur Spielfläche, die Seiten an den Ecktaschen einen Winkel von 35° und an den Seitentaschen einen Winkel von 70°. Neben den Banden wurden auf dem Tisch zwischen den Taschen jeweils 3 Zielhilfen platziert, die aus Kegeln mit vier Seiten und einer Höhe von 0.25 bestehen.

Die Standfüße des Tische bestehen aus einfachen Quadern, deren Kanten mit einem Bevel abgerundet wurden. Der Tisch selbst steht auf einer Ebene, Seitenwände und Decke gibt es in der Szene nicht.

Die genauen Maße des Tischs sind in den Abbildungen 4, 5 und 6 noch einmal übersichtlich dargestellt, der fertige Tisch ist in Abbildung 7 zu sehen.

Die Texturierung des Tischs verlief etwas schwieriger, da bisher noch keine Erfahrungen mit der Texturierung von Objekten in *Maya* vorhanden waren. Dennoch wurden die angestrebten Ziele erreicht. Die Banden sowie das Spielfeld erhielten eine grüne Stofftextur (Abbildung 8), die möglichst klein gewählt und gekachelt auf die Objekte gelegt wurde. Der Boden besitzt eine Steintextur, die nach außen hin schwarz wird. Um später eine aufwändige Berechnung des Tischschattens zu ersparen, wurde der Schatten des Tischs direkt in die Textur integriert (Abbildung 9). Der restliche Tisch besitzt, genau wie die Zielhilfen, ein einfaches Lambert Material ohne Textur.

3.2 Queue und Kugeln

Eine separate Modellierung der Kugeln und des Queues war nicht notwendig, da es sich hierbei um geometrische Primitive handelt und OpenGL bereits Befehle enthält, um diese zu erzeugen. Für die Realisierung wurden sogenannte *Quadric Objects* verwendet. *Quadric Objects* haben im Gegensatz zu beispielsweise *glutSolidSphere* den Vorteil mit Texturkoordinaten erzeugt zu werden. Die Erstellung dieser Objekte erfordert allerdings etwas mehr Co-

[1] Bevel: Rundet die Kanten eines Objekts ab.

[2] Boolean-Operation: Ein Objekt wird vom anderen "abgezogen", so können Hohlräume in Objekte geschnitten werden.

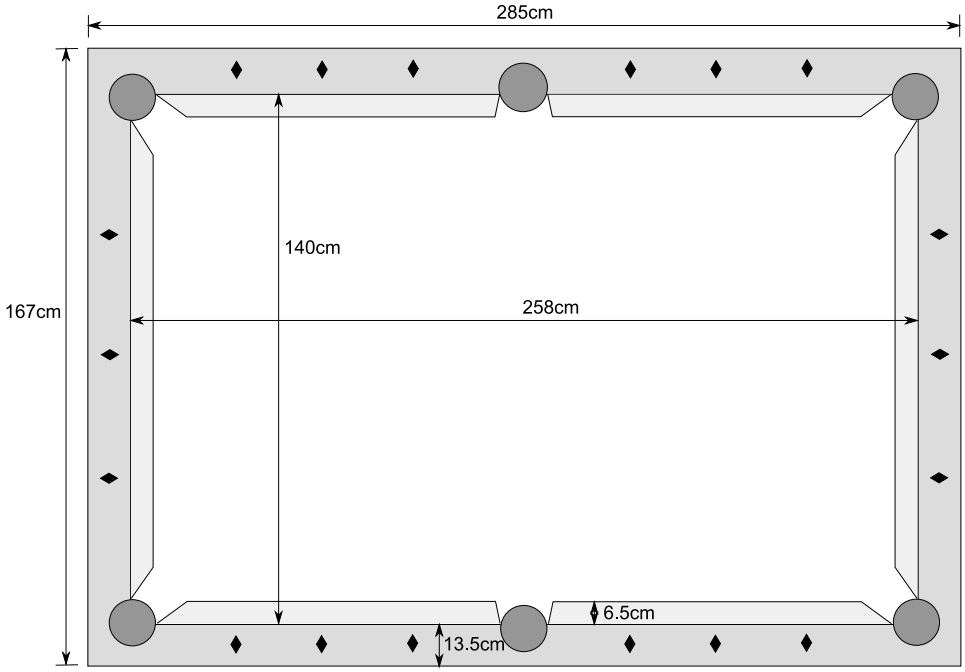

Abbildung 4: Die genauen Maße des Tischs.

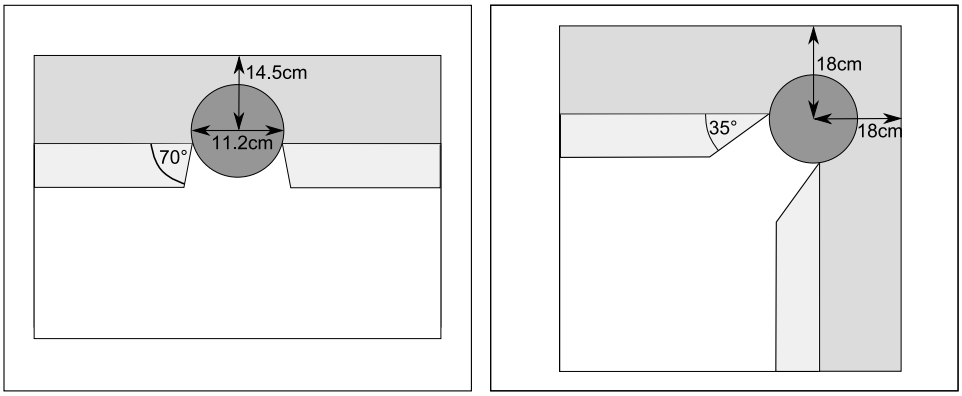

Abbildung 5: Links: Maße an den Seitentaschen. Rechts: Maße an den Ecktaschen.

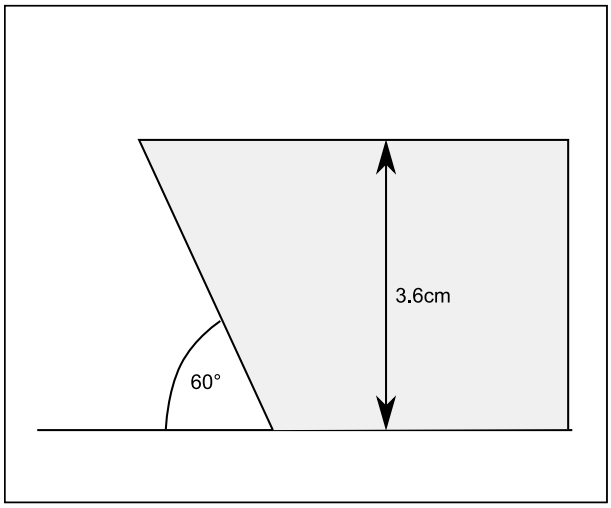

Abbildung 6: Querschnitt durch eine Bande.

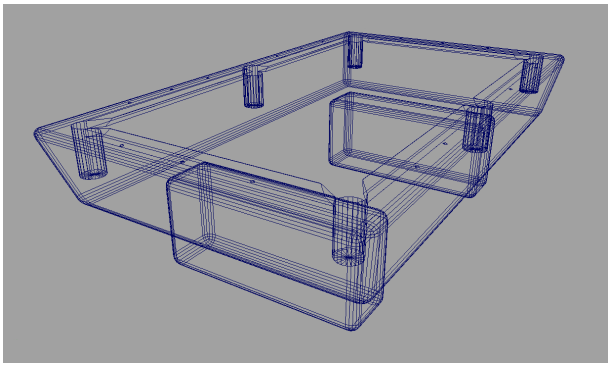

Abbildung 7: Drahtgitterdarstellung des fertigen Tischmodells.

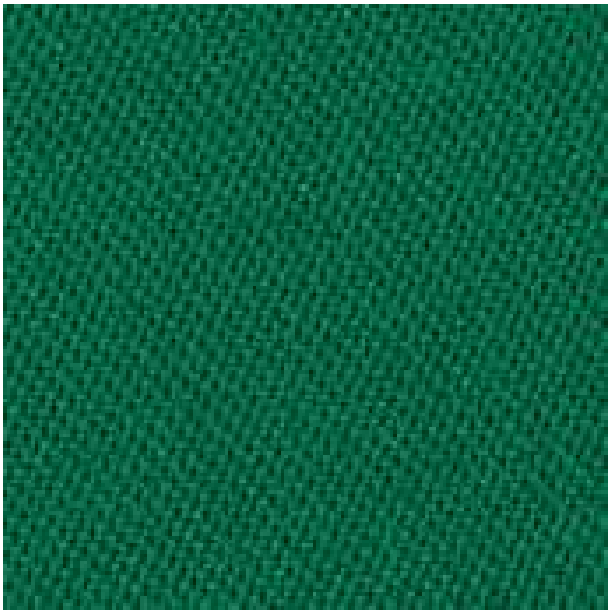

Abbildung 8: Textur des Spielfelds.

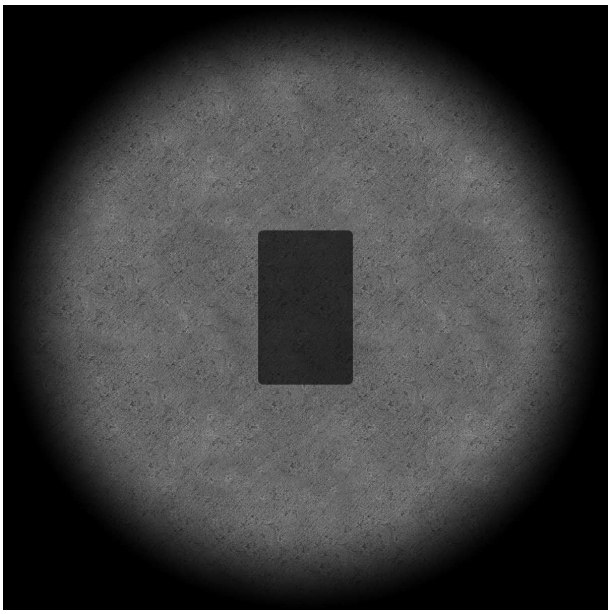

Abbildung 9: Textur des Bodens mit eingezeichnetem Schatten für den Tisch.

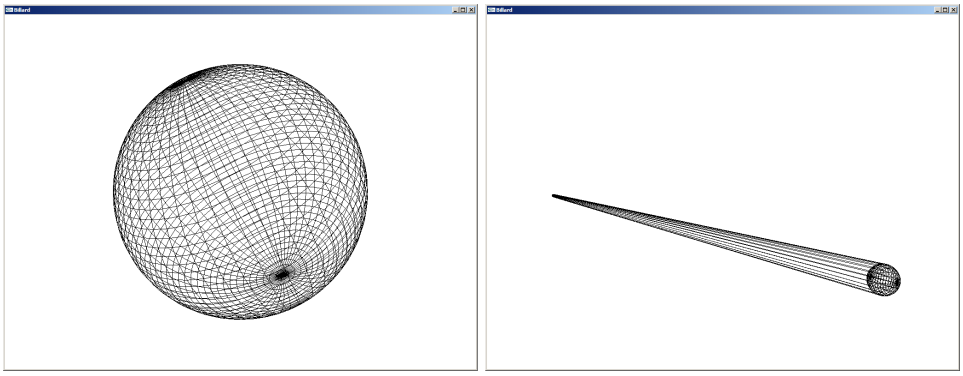

Abbildung 10: Drahtgitterdarstellung der Modelle für eine Kugel und den Queue.

Abbildung 11: Textur des Queues.

de als bei der Verwendung von *glutSolidSphere*. Dieser wird in Kapitel 4.5.2 genau erläutert. Abbildung 10 zeigt das fertige Ergebnis.

Die weiße Kugel sowie die Kugel, die den Knauf des Queues bildet, tragen einfache einfarbige Texturen. Die Textur des Queues (Abbildung 11) wurde aus zwei verschiedenen Holztexturen zusammengesetzt und mit einer blauen Spitze sowie zwei schwarzen Ringen in der Mitte des Queues ergänzt. Die Texturen für die Kugeln wurden so erstellt, dass die Nummern der Kugeln auf zwei gegenüberliegenden Seiten des Objekts gut erkennbar sind (Abbildung 12). Wichtig ist hierbei, die Texturen abschließend um ca. 50% horizontal zu stauchen, da diese etwas verzerrt auf die Kugeln gelegt werden, wie in Abbildung 13 gut erkennbar ist.

4 Implementierung

Die Implementierung nahm bei dieser Arbeit den größten Anteil ein. Aus diesem Grund werden in diesem Kapitel die grundlegenden Implementierungsschritte hergeleitet sowie die neuen Klassen und ihre Beziehungen zueinander vorgestellt. Ebenso wird auf die physikalischen Berechnungen und die Darstellung des Spiels eingegangen.

4.1 Klassen

Für die Realisierung des Spiels wurden verschiedene Klassen entworfen, die für die Verwaltung und Berechnung der verschiedenen Objekte zuständig sind. Die folgenden Unterkapitel erklären die wichtigsten Klassen des Pro-

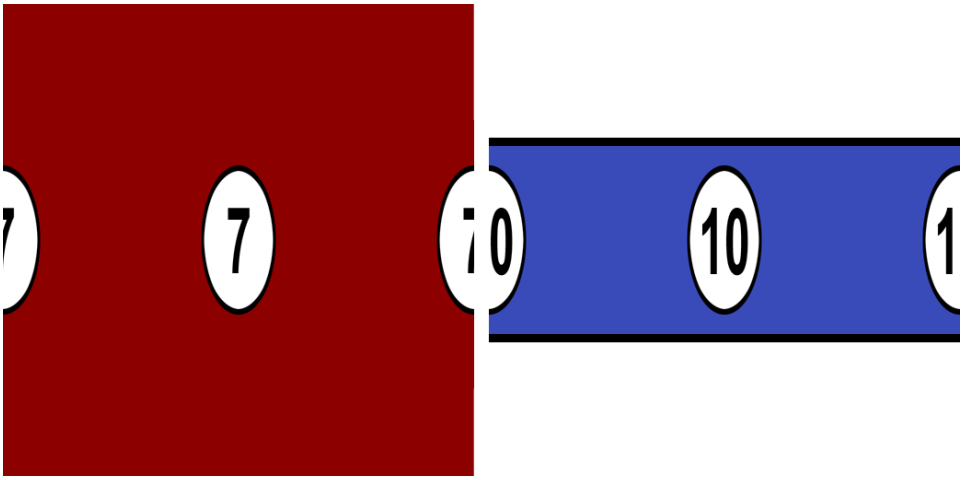

Abbildung 12: Texturen der Kugeln. Links eine volle Kugel, rechts eine halbe.

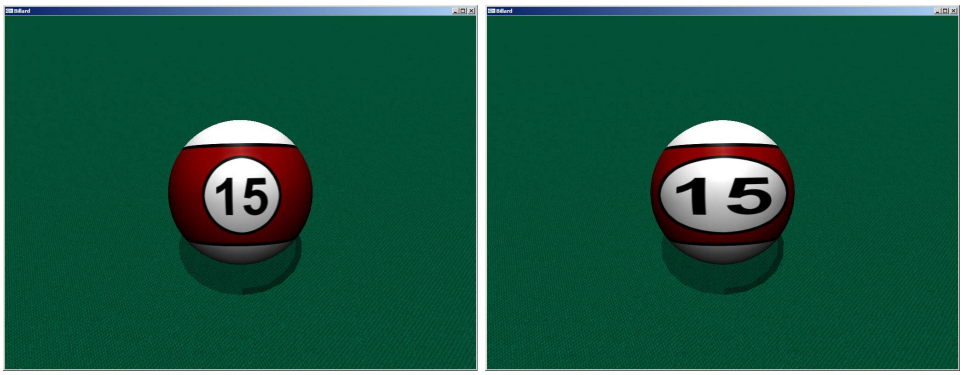

Abbildung 13: Vergleich zwischen einer gestauchten (links) und einer ungestauchten Textur (rechts).

jekts. Das Klassendiagramm in Abbildung 14 verdeutlicht die Beziehung der Klassen untereinander.

4.1.1 Ball

Die Kugeln wurden als Objekte vom Typ *Ball* realisiert, sodass für jede Kugel jederzeit alle nötigen Informationen verfügbar sind. Jede Kugel besitzt eine fortlaufende Nummer von 0 bis 15, wobei die weiße Kugel die Nummer 0 und alle anderen Kugeln die Nummer entsprechend ihrer Textur besitzen. Außerdem besitzt jede Kugel eine Masse von 170g, was gültigen Turniernormen entspricht. Die Position und die aktuelle Geschwindigkeit der Kugel wird als *Vec3D* Objekt realisiert, die aktuelle Orientierung der Kugel als *Quaternion*. Um zu bestimmen, ob eine Kugel bereits eingelocht wurde und somit nicht mehr gezeichnet werden muss, gibt es eine Bool-Variable. Damit

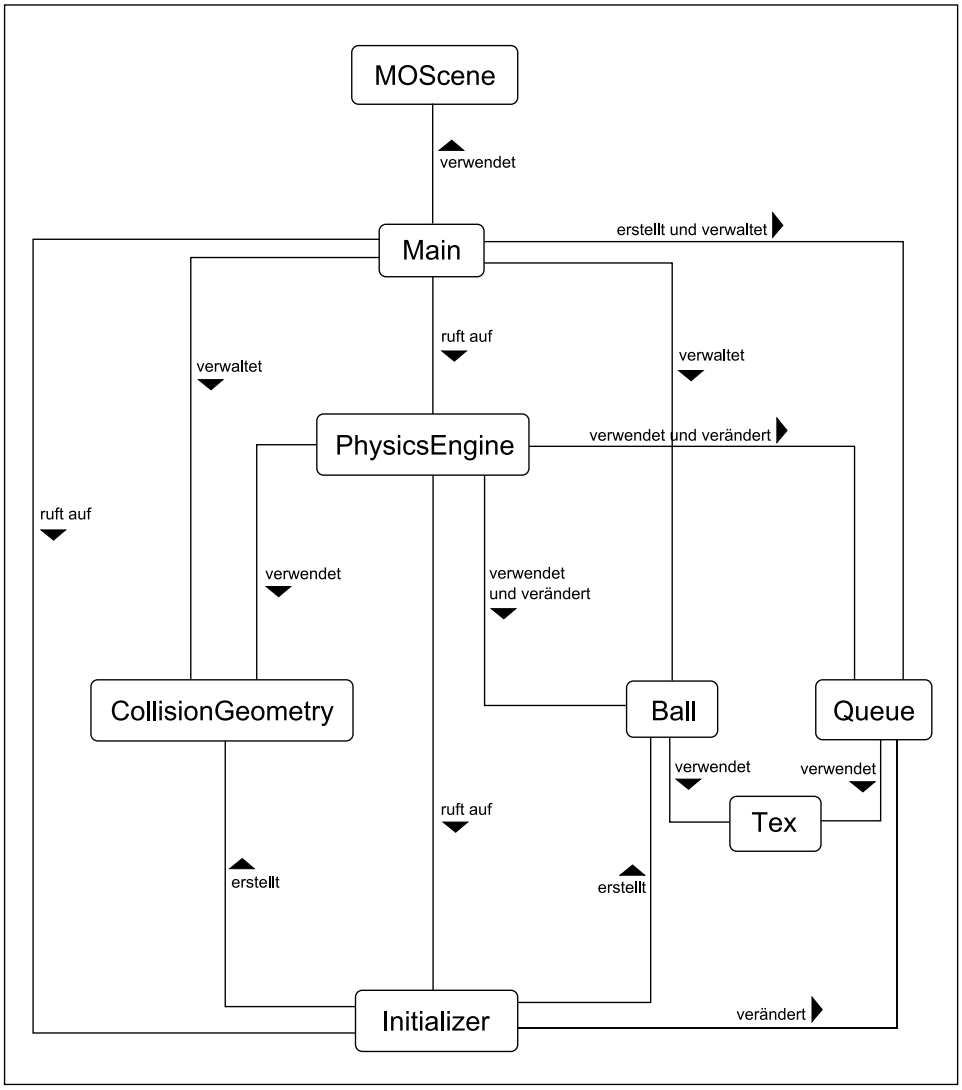

Abbildung 14: Das Klassendiagramm verdeutlicht die Beziehung der einzelnen Klassen untereinander.

der Spieler die Kugeln unterscheiden kann, besitzt jede Kugel eine Textur.

Die Klasse *Ball* besitzt zwei Methoden, *draw* und *randomizeOrientation*. Letztere erzeugt lediglich eine zufällige Orientierung der Kugel, wenn das Spiel initialisiert bzw. zurückgesetzt wird. Dazu wird ein *Quaternion* mit vier Zufallszahlen initialisiert und anschließend normalisiert. In der *draw* Methode wird die Kugel mit ihrer Orientierung an ihrer aktuellen Position gezeichnet.

4.1.2 CollisionGeometry

Alle Flächen, mit denen später die Kugeln kollidieren können, sind Objekte vom Typ *CollisionGeometry*. Die Klasse speichert für jedes Objekt drei Eckpunkte sowie die normalisierte Normale. Zudem besitzt sie eine Variable *isBouncing*, die angibt, ob die Kugel von diesem *CollisionGeometry* Objekt abprallt oder nicht, also ob es sich um eine Bande oder die Außenbegrenzung einer Tasche handelt. Des weiteren besitzt jedes *CollisionGeometry* Objekt eine ID zur eindeutigen Identifizierung.

Außerdem besitzt sie eine *draw* sowie eine *drawNormals* Methode. In der *draw* Methode wird das Objekt dargestellt, die *drawNormals* Methode zeigt zusätzlich die Normale des Objekts mit Hilfe eines roten Strichs an. Die Kollisionsgeometrie ist normalerweise nicht sichtbar, die Methoden *draw* und *drawNormals* dienen lediglich der Visualisierung zu Testzwecken.

4.1.3 Initializer

Die Klasse Initializer dient dem Initialisieren der Kugeln und der Kollisionsgeometrie sowie dem Zurücksetzen des Spiels. Dem entsprechend besitzt sie auch nur drei Methoden, *initBalls*, *initRails* und *resetGame*. In der Methode *initBalls* werden die Kugeln auf ihren Startpositionen und mit der passenden Textur erzeugt. In der Realität liegen die Kugeln nie absolut aneinander, sodass die Kugeln auch im Spiel nicht immer an exakt derselben Position liegen sollten. Aus diesem Grund liegen die Startpositionen der Kugeln um $2r + 0.2$ auseinander, sodass auf die X und Z-Koordinate eine Zufallszahl zwischen -0.1 und 0.1 aufaddiert werden kann. Somit gleicht kein Anstoß dem anderen. Die Methode *initRails* erstellt alle Dreiecke der Kollisionsgeometrie mit ihrer Normalen, die Methode *resetGame* setzt alle Objekte wieder in ihre Ausgangsposition beim Start des Spiels zurück.

4.1.4 PhysicsEngine

Die PhysicsEngine bildet das Herzstück des Programms. Hier wird mit der Methode *calculatePhysics* 250 mal pro Sekunde eine Physikberechnung durchgeführt, in der die Geschwindigkeit jeder Kugel, eine mögliche Kollision mit anderen Kugeln oder einer Bande sowie die aktuelle Orientierung jeder Kugel berechnet wird. Außerdem wird hier getestet, ob eine Kugel gerade versenkt

wurde. Für eine bessere Übersicht wurden die Kollisionstests in die Methoden *calculateBallCollision* und *calculateRailCollision* und der Test zum Versenken einer Kugel in die Methode *checkForPockets* ausgelagert.

4.1.5 Queue

Für den Queue wurde ebenfalls ein eigenes Objekt angelegt, sodass alle Informationen, die den Queue betreffen, ausgelagert werden konnten. Der Queue besitzt zunächst einmal eine Position, die sich allerdings nicht als absolute Position, sondern als den Versatz der Queuespitze zum Mittelpunkt der weißen Kugel versteht. Eine absolute Position ist nicht notwendig, da die Orientierung des Queues über die Kameraposition und die Queueneigung, für die es im Queue ebenfalls einen separaten Wert gibt, errechnet wird. Des weiteren gibt es noch zwei Variablen für den Versatz des Queues relativ zum Mittelpunkt der weißen Kugel in XZ und Y-Richtung, die ein dezentrales Anspielen der weißen Kugel ermöglichen. Über die Werte dieser beiden Variablen ist es später möglich, die Laufrichtung der weißen Kugel nach einer Kollision zu verändern. Außerdem benötigt der Queue zwei Variablen für die benötigten Texturen.

Auch der Queue besitzt eine *draw* Methode. Darin wird der Queue zunächst entsprechend der Kameraposition und der Neigung des Queues rotiert und verschoben. Der Queue selbst besteht wiederum aus zwei *Quadric Objects*, einem Zylinder, der an einem Ende einen geringeren Durchmesser als am anderen Ende hat, und einer Kugel, die den Queue abschließt.

4.1.6 Quaternion, Tex, Vec3D

Die Klasse *Quaternion* stellt sämtliche Methoden bereit, um Quaternionen zu erzeugen, zu verändern und mit ihnen zu rechnen. Dies ist ausschließlich für die Berechnung der Orientierung der Kugeln nötig. Quaternionen bestehen aus einer Variablen w, die den Rotationswinkel angibt, sowie einem Vector $\vec{v}$, der die Rotationsachse bestimmt. Die Klasse besitzt Methoden zum Addieren und Multiplizieren von Quaternionen, sowie zum Normalisieren eines Quaternions.

Die Klasse *Tex* ist für das Laden und Aktivieren von Texturen zuständig. Mittels *glAux* können BMP-Dateien geladen werden. Die Methode *useTexture* sorgt dafür, dass die Textur aktiviert wird.

Die Klasse *Vec3D* stellt Methoden bereit, um dreidimensionale Vektoren zu erzeugen und mit ihnen zu rechnen. Jedes *Vec3D* Objekt besitzt drei float-Variablen, x, y und z.

4.2 Main-Loop

In *GLUT-Programmen* läuft die *display* Methode immer in einer Endlosschleife. Diese Methode wurde genutzt, um alle nötigen Programmteile auf-

zurufen. Der Reihe nach wird hier die Kamera gesetzt, die Geometrie gezeichnet, die Physikberechnungen durchgeführt sowie die Framerate berechnet.

Zunächst wird die aktuell vergangene Zeit seit Programmstart mittels *glutGet(GLUT_ELAPSED_TIME)* ermittelt und gespeichert. Danach wird die Kamera mittels *gluLookAt* gesetzt. Anschließend werden der Tisch, die Kugeln, der Queue und, wenn aktiviert, die Kollisionsgeometrie gezeichnet. Nun wird, falls die Simulation der Physik mit einem Anstoß gestartet wurde, die Methode *calculatePhysics* der PhysicsEngine aufgerufen. Zum Schluss wird nochmals die Zeit ermittelt, woraus sich dann die aktuelle Framerate mit Formel 4 errechnen lässt:

$$f = \frac{1000}{t_{end} - t_{start}} \tag{4}$$

4.3 Physik

Da die Physikberechnung das Kernstück dieser Arbeit darstellt, soll an dieser Stelle näher darauf eingegangen werden. So sollen die zu Beginn auf dem Tisch befindlichen 15 bzw. 9 Kugeln alle miteinander interagieren können. Ebenso muss auch das Zusammenspiel von Kugeln mit Banden und Taschen des Tischs betrachtet werden. Hierfür wird im folgenden Kapitel die Implementierung der Kollisionsbehandlung, das Einlochen der Kugeln und der Effet näher erklärt.

Zunächst wird festgelegt, wie oft die Physik pro Frame laufen muss, um insgesamt 250 mal pro Sekunde berechnet zu werden. Zu diesem Zweck wird der Methode *calculatePhysics* die aktuelle Framerate übergeben und so die Anzahl der Schleifendurchläufe ermittelt.

In jedem Berechnungsschritt läuft ein Iterator über alle Kugeln. Für jede Kugel werden folgende Berechnungen durchgeführt:

1. Berechnung der neuen Position
2. Berechnung der neuen Orientierung
3. Kollisionstest mit den Banden
4. Prüfen, ob die Kugel versenkt wurde
5. Kollisionstest mit allen anderen Kugeln
6. Berechnung der neuen Geschwindigkeit

In diesem Kapitel wird oft von Geschwindigkeiten gesprochen. Wichtig zu erwähnen ist, dass sich diese Geschwindigkeiten dreidimensional verstehen. Die Geschwindigkeit einer Kugel ist als *Vec3D* Objekt realisiert und gibt neben der eigentlichen Geschwindigkeit, dem Betrag dieses Vektors, auch die Bewegungsrichtung an.

4.3.1 Ballphysik

Die Ballphysik ist ein elementarer Bestandteil der Physikengine. Unter Ballphysik wird die Bewegung der Kugeln auf ihrer zugewiesenen Laufbahn verstanden. Dazu gehört zum einen das Verhalten der Kugel direkt nach dem Anstoß, wie schnell sie sich in welche Richtung bewegt. Zum anderen gehört dazu das Verhalten in der Bewegung. Wie wird die Kugel durch Reibung mit der Spielfläche abgebremst und wie rotiert sie um die eigene Achse. Wie verhält sich die Kugel bei einer Kollision mit einer anderen Kugel oder der Bande, und wie wirkt sich der Effet aus. Dies wird im Folgenden genau erläutert.

Wird die weiße Kugel angestoßen, beginnt sie zu rollen. Der Anstoß sowie die Berechnung der Anfangsgeschwindigkeit der Kugel wird später in Kapitel 4.4.3 erklärt. Solange sie kein anderes Objekt trifft, hält dieser Zustand an, die Geschwindigkeit der weißen Kugel verringert sich allerdings mittels Formel 5.

$$\vec{v}_{new} = \vec{v}_{old} \cdot 0.9965 \tag{5}$$

Nähert sich die Summe der Geschwindigkeiten aller Kugeln dem Wert 0, sodass keine sichtbare Bewegung der Kugeln mehr stattfindet, wird die Geschwindigkeit jeder Kugel auf 0 gesetzt, die neue Kameraposition berechnet und der Queue auf seine Standardposition zurückgesetzt. Die neue Kameraposition berechnet sich über die alte Position, auf die der Vektor von der Position der weißen Kugel vor dem Stoß zur Position nach dem Stoß addiert wird. Wurde bei diesem Stoß die weiße Kugel versenkt, wird sie auf die Startposition gesetzt. Wurde die schwarze Kugel versenkt, wird das komplette Spiel zurückgesetzt, das heißt alle Kugeln sowie Queue und Kamera auf die Startposition gesetzt.

Trifft die Kugel auf eine Bande, wird die Kollision behandelt, die neue Richtung berechnet und die Geschwindigkeit mit 0.9 multipliziert, da bei einer Kollision immer Kraft verloren geht. Trifft die Kugel auf eine andere Kugel, wird ebenfalls die Kollision behandelt und die neue Richtung sowie Geschwindigkeit beider Kugeln berechnet und mit 0.95 multipliziert, da auch hier Kraft verloren geht. Kollisionserkennung und -behandlung werden in den folgenden Kapiteln genauer erklärt.

Läuft die Kugel in eine Tasche, wird über einen Test erkannt, dass die Kugel versenkt wurde. Nun wird der Kugel eine Geschwindigkeit in negativer Y-Richtung zugewiesen, die Kugel beginnt also zu fallen. Unterschreitet sie dabei den Wert -20, so wird sie nicht mehr gerendert. Wie diese Prozedur genau abläuft, wird in Kapitel 4.3.5 genau erläutert.

Die Kugeln sollen im Spiel auch richtig rollen. Dieses Rollen ist nichts anderes, als ein ständiges Rotieren der Kugel in die gewünschte Richtung. Wie weit rotiert werden muss, lässt sich, wie in Abbildung 15 dargestellt, über die zurückgelegte Strecke d sowie den Radius r der Kugel mittels Formel

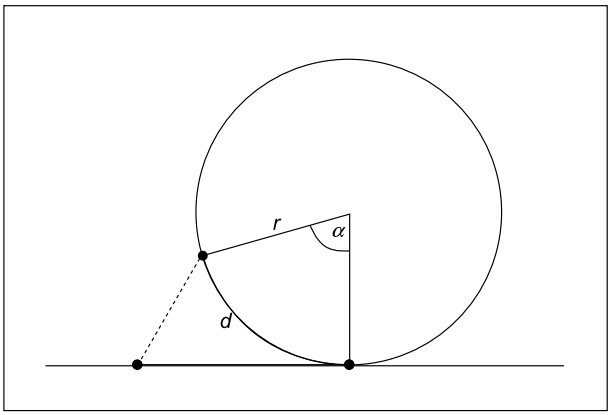

Abbildung 15: Berechnung des Rotationswinkels α über die zurückgelegte Distanz d und den Radius r.

6 bestimmen.

$$\alpha = \frac{d}{r} \tag{6}$$

Für eine Rotation wird nun noch die Rotationsachse benötigt. Diese lässt sich über das Kreuzprodukt zwischen einem Vektor parallel zur Y-Achse und der Bewegungsrichtung $\vec{d}$ mit Formel 7 errechnen.

$$\vec{a} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \times \vec{d} \tag{7}$$

Die einfachste Möglichkeit wäre, die Rotation als einfache *Axis-Angle Rotation* durchzuführen. Eine *Axis-Angle Rotation* beschreibt die Rotation eines Objekts in einem Winkel um eine Rotationsachse. Das heißt in jedem Bewegungsschritt wird der Rotationswinkel berechnet und auf den alten Winkel aufaddiert, um die gesamte Rotation darzustellen. Bei der Darstellung würde die Kugel dann lediglich in diesen Winkel um die berechnete Rotationsachse gedreht. So lange die Kugel geradeaus rollt, funktioniert diese Variante auch sehr gut. Kollidiert die Kugel aber mit einem anderen Objekt, findet ein abrupter Richtungswechsel statt. Die Rotationsachse ändert sich bei einer Kollision schlagartig. Da die alte Rotationsachse bei der Berechnung aber nicht mit einbezogen werden kann, "springt" die Orientierung der Kugel schlagartig, wie in Abbildung 16 gut zu erkennen ist.

Ein weiterer Nachteil ist die Genauigkeit der Berechnung der Rotationsachse. Je geringer die zurückgelegte Distanz ist, desto ungenauer wird das Ergebnis $\vec{a}$. Das führt bei langsamen Kugeln zu einem "Wackeln".

Diese Nachteile schließen eine einfache *Axis-Angle Rotation* für dieses Spiel aus. Stattdessen wurde eine Rotation mit Hilfe von Quaternionen implementiert. Quaternionen können jede beliebige Orientierung eines Objekts

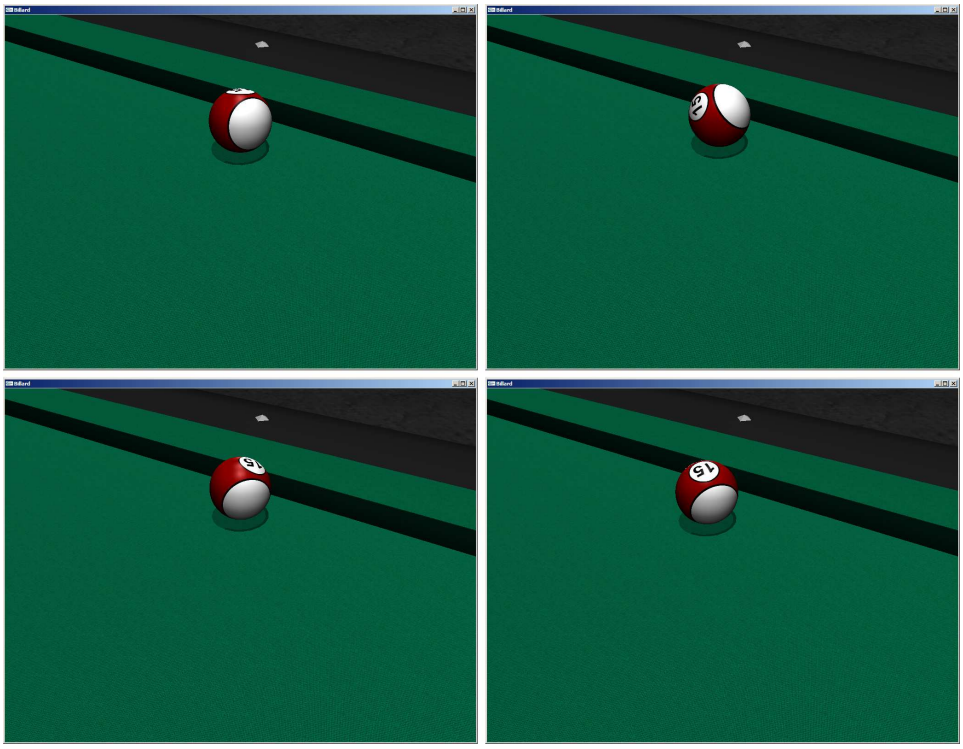

Abbildung 16: Vergleich zwischen *Axis-Angle Rotation* (oben) und Quaternionen (unten). Links ist jeweils der letzte Schritt vor und rechts der erste Schritt nach der Kollision zu sehen. Bei der *Axis-Angle Rotation* "springt" die Orientierung der Kugel, bei der Rotation mittels Quaternionen ändert sich die Orientierung korrekt.

im dreidimensionalen Raum beschreiben, ähnlich der *Axis-Angle Rotation* über einen Winkel und eine Rotationsachse. Der große Vorteil liegt aber darin, dass Quaternionen miteinander multipliziert werden können und durch das Ergebnis mehrere hintereinander durchgeführte Rotationen um verschiedene Achsen in einer Rotation um eine Achse ausgedrückt werden können. Es geht also bei einem abrupten Richtungswechsel im Gegensatz zur einfachen *Axis-Angle Rotation* keine Information verloren, es kommt nur eine neue hinzu.

Ein Quaternion besteht aus zwei Komponenten, wie in Formel 8 dargestellt. Zum einen aus einer Zahl, die später den Drehwinkel darstellt, zum anderen aus einem Vektor, der die Rotationsachse angibt (Quelle: [13]).

$$q = (w, \vec{z}) \tag{8}$$

Ein Quaternion, das eine Drehung um den Winkel α um die Achse $\vec{a}$ darstellt,

Listing 1: Berechnung der Orientierung einer Kugel mit Hilfe von Quaternionen

```
1  Vec3D direction = newBallPosition - oldBallPosition;
2  if (length(direction) > 0.000001)
3  {
4     Vec3D rotationAxis = cross(Vec3D(0, 1, 0), direction);
5     rotationAxis  = normalize(rotationAxis);
6     float rotationAngle = length(direction) / 2.86;
7     Quaternion quat;
8     quat[W] = cos(rotationAngle / 2);
9     quat[Z] = sin(rotationAngle / 2) * rotationAxis;
10    Quaternion newOrientation;
11    newOrientation = quat * oldOrientation;
12 }
```

ist in Formel 9 zu sehen.

$$q = (\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2} \cdot \vec{a}) \tag{9}$$

Auf diese Weise lässt sich bequem von einer *Axis-Angle Rotation* in ein Quaternion und wieder zurück rechnen. Dies ist nötig, da die Rotation der Kugel in einem Bewegungsschritt zunächst als *Axis-Angle Rotation* berechnet und danach auf die alte Orientierung der Kugel aufaddiert wird. Für die spätere Darstellung wird die Orientierung der Kugel wieder in eine *Axis-Angle Rotation* zurückgerechnet, um die Kugel mittels *glRotatef* rotieren zu können. Das bereits angesprochene Multiplizieren zweier Quaternionen, um deren Rotationen in einem Quaternion zu vereinen, ist in Formel 10 dargestellt.

$$q = q_1 \cdot q_2 = (w_1 \cdot w_2 - \vec{z_1} \circ \vec{z_2}, w_1 \cdot \vec{z_2} + w_2 \cdot \vec{z_1} + \vec{z_1} \times \vec{z_2}) \tag{10}$$

Damit lässt sich die Orientierung jeder Kugel genau bestimmen. Listing 1 zeigt die Implementierung dieser Berechnungen.

4.3.2 Effet

Eine große Herausforderung stellte die Implementierung des Effets dar, da eine physikalisch korrekte Umsetzung den Zeitraum dieser Arbeit überschritten hätte. Aus diesem Grund wurde ein physikalisch nicht korrekter, aber spielerisch dennoch ansprechender Ansatz gewählt. Die Grundidee bestand darin, nur einfachen Effet zu implementieren. Die bei den Grundlagen erwähnten *Massestöße* sind dabei nicht möglich. Die Stoßkugel kann zwar überall angespielt werden, läuft aber immer geradeaus. Sobald eine Kollision mit einem anderen Objekt stattfindet, wird die neue Richtung der Kugel entsprechend dem Effet verändert. Bei einer Kollision mit einer Bande gibt es grundsätzlich zwei Fälle. Es muss unterschieden werden, ob die Kugel von links auf die Bande trifft, oder von rechts. Wurde die Kugel links angespielt

und trifft von links auf die Bande, so verkleinert sich der Ausfallswinkel, trifft sie von rechts auf die Bande vergrößert er sich. Wurde die Kugel rechts angespielt, verläuft die Berechnung analog. Wurde die Kugel oben oder unten angespielt, wird der Effet nur verringert und invertiert.

Bei der Kollision zweier Kugeln läuft die Berechnung etwas anders ab. Der Effet wirkt hier nur, wenn die Kugel oben oder unten angespielt wurde. Wurde die Stoßkugel oben angespielt, rollt sie der getroffenen Kugel hinterher. Wurde sie unten angespielt, rollt sie wieder zurück.

Die genaue Effetberechnung wird in den folgenden beiden Kapiteln zusammen mit der Kollisionserkennung und Behandlung erläutert. Zu erwähnen ist, dass der Effet nur für die Stoßkugel berechnet wird.

4.3.3 Kollision zwischen zwei Kugeln

Trifft eine Kugel auf eine andere Kugel, so ändert sich das Bewegungsverhalten beider Kugeln. Wurde die Kugel mit Effet angespielt, muss auch dieser Effekt bei der Berechnung berücksichtigt werden. Wie die Kollision zweier Kugeln erkannt und die neuen Bewegungsrichtungen berechnet werden, wird in diesem Kapitel erläutert.

Für die Kollisionsberechnung zwischen den Kugeln gibt es zwei Möglichkeiten. Zum einen kann die komplette Laufbahn aller Kugeln vorberechnet und der früheste Schnittpunkt ermittelt werden. Danach läuft die Simulation so lange, bis der Zeitpunkt der ersten Kollision erreicht ist. Daraufhin wird die Kollision behandelt und danach wieder die Laufbahn der Kugeln berechnet. Diese Variante besitzt natürlich den Vorteil, dass kein Schnittpunkt übersehen wird. Allerdings besitzt sie darüber hinaus mehr Nachteile als Vorteile. Jede Kugel benötigt zwei Strahlen, die erst berechnet und dann jeweils untereinander geschnitten werden müssen, das entspricht bei 16 Kugeln 480 Schnitttests zwischen Geraden. Des Weiteren kann diese Methode nur gerade Kugellaufbahnen effizient berechnen, Schnitttests zwischen Kurven sind deutlich aufwändiger.

Bei der zweiten Möglichkeit wird bei jeder Physikberechnung der Abstand der Kugeln untereinander gemessen. Ist der Abstand zweier Kugeln kleiner als 2r, hat eine Kollision stattgefunden und kann behandelt werden. Der Vorteil dieser Methode ist die effiziente Berechnung, da lediglich 120 mal der Abstand zweier Punkte berechnet werden muss. Außerdem macht es keinen Unterschied, ob die Kugeln geradeaus oder Kurven laufen, die Berechnung bleibt hierbei dieselbe. Der große Nachteil dieser Methode liegt allerdings in ihrer Abhängigkeit von der Berechnungsrate. Wird die Physik zu selten berechnet, können unter Umständen Kollisionen übersehen werden, wird sie zu oft berechnet, bricht die Performance ein.

Obwohl dieses Programm die Physikberechnungen zunächst nur im zweidimensionalen Raum durchführen sollte, springende Kugeln sowie Reibung also zunächst vernachlässigen sollte, fiel die Wahl auf die zweite Möglichkeit.

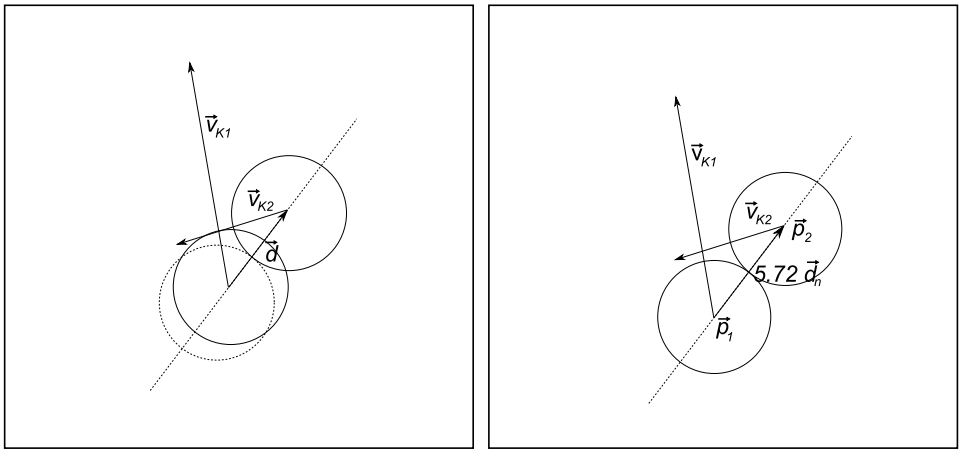

Abbildung 17: Links: zwei Kugeln durchdringen sich bei der Kollision. Rechts: die erste Kugel wurde zurückgesetzt, um eine Überschneidung zu verhindern.

So steht einer Erweiterung auf den dreidimensionalen Raum und der Implementierung von Reibung, sodass die Kugeln auch Kurven laufen können, nichts im Wege.

Der Methode *calculateBallCollision* werden zunächst die beiden zu testenden Kugeln sowie der Queue übergeben. Im Queue ist der Anspielpunkt auf der weißen Kugel gespeichert, der für die Berechnung des Effets wichtig ist. Danach wird der Vektor vom Mittelpunkt der ersten zum Mittelpunkt der zweiten Kugel, $\vec{d}$, sowie dessen Länge bestimmt. Ist die Länge kleiner als 5.72, was $2r$ entspricht, so hat eine Kollision stattgefunden und kann behandelt werden.

Da sich die Kugeln bei dieser Methode zwangsweise durchdringen und sich aufgrund der Art der Berechnung unter gewissen Umständen verhaken können, werden die Kugeln zunächst wieder auseinander gesetzt, wie in Abbildung 17 verdeutlicht. Dazu wird $\vec{d}$ zunächst normalisiert und die Position der ersten Kugel mit Formel 11 berechnet.

$$\vec{p_1} = \vec{p_2} - \vec{d_n} \cdot 5.72 \tag{11}$$

Nachdem sich die Kugeln nun nicht mehr überschneiden, wird eine Normale zu $\vec{d}$, $\vec{n}$, in der XZ-Ebene berechnet. Jetzt können die neuen Geschwindigkeiten der beiden Kugeln berechnet werden, wie in Abbildung 1 dargestellt. Dazu werden die Geschwindigkeiten der beiden Kugeln in zwei Komponenten aufgeteilt, eine $\vec{v}_d$ parallel zu $\vec{d}$ und eine $\vec{v}_n$ parallel zu $\vec{n}$.

$\vec{v}_n$ und $\vec{v}_d$ lassen sich berechnen mit den Formeln 12 bis 15 (Quelle: [6]).

$$v_{d_{K1}} = \vec{v}_{K1} \circ \vec{d_n} \tag{12}$$

$$v_{n_{K1}} = \vec{v}_{K1} \circ \vec{n} \tag{13}$$

$$v_{d_{K2}} = \vec{v}_{K2} \circ \vec{d}_n \tag{14}$$

$$v_{n_{K2}} = \vec{v}_{K2} \circ \vec{n} \tag{15}$$

Nach dieser Aufteilung können die neuen Geschwindigkeiten der Kugeln berechnet werden mit den Formeln 16 und 17

$$\vec{v}_{K1} = (v_{d_{K2}} \cdot \vec{d}_n + v_{n_{K1}} \cdot \vec{n}) \cdot 0.95 \tag{16}$$

$$\vec{v}_{K2} = (v_{d_{K1}} \cdot \vec{d}_n + v_{n_{K2}} \cdot \vec{n}) \cdot 0.95 \tag{17}$$

Die Skalierung mit dem Faktor 0.95 findet statt, weil in der Realität auch immer etwas Kraft bei einer Kollision verloren geht.

Bei der weißen Kugel liegt ein Sonderfall vor, da noch die Modifikatoren für den Effet hinzukommen. Wie bereits in Kapitel 4.3.2 erläutert, wirkt sich bei der Kollision zweier Kugeln nur der Effet in oder entgegen der Laufrichtung der Kugel aus. Zunächst wird der Geschwindigkeitsfaktor f aus der Geschwindigkeit der Kugel berechnet. Dieser ist nötig, da die Kugel bei stärkerem Anspielen einen größeren Effet besitzt als bei leichtem Anstoßen. Der Faktor f berechnet sich mit Formel 18. Die Division durch 4 hat sich in zahlreichen Tests als der Skalierungsfaktor mit dem besten Spielgefühl bewiesen.

$$f = \frac{|\vec{v}_{K1}|}{4} \tag{18}$$

Um keinen unrealistisch starken Effet zu erzeugen, wird f auf einen Maximalwert von 100 beschränkt. Anschließend wird der Effet auf die neue Geschwindigkeit der weißen Kugel aufaddiert. Wenn e_y der Effet in Y-Richtung ist, siehe Kapitel 4.4.4, berechnet sich die neue Geschwindigkeit $\vec{v}_e$ aus der alten bereits berechneten Geschwindigkeit $\vec{v}$, der Geschwindigkeit vor der Kollision $\vec{v}_{old}$ und f mit Formel 19.

$$\vec{v}_e = \vec{v} + \frac{\vec{v}_{old}}{|\vec{v}_{old}|} \cdot e_y \cdot f \tag{19}$$

Nach der ersten Kollision geht ein Großteil des Effets verloren, daher wird er mit Faktor 0.2 skaliert. Ist $e_y < 0$, wurde die weiße Kugel also unten angespielt, so wird der Effet nun invertiert, damit die weiße Kugel nach der ersten Kollision, die mit einem rückwärtigen Effet behandelt wurde, einen vorwärtigen Effet erhält.

4.3.4 Kollision zwischen Kugel und Bande

Trifft eine Kugel auf eine Bande, prallt sie von dieser ab. Um eine Kollision mit der Bande zu erkennen, ist ein Schnitttest zwischen der Kugellaufbahn und der Bande nötig. Wurde ein Schnittpunkt gefunden und hat die Kugel die Bande schon durchdrungen, müssen die neue Laufrichtung und Geschwindigkeit der Kugel berechnet werden. Wie dies genau abläuft, wird im Folgenden erklärt.

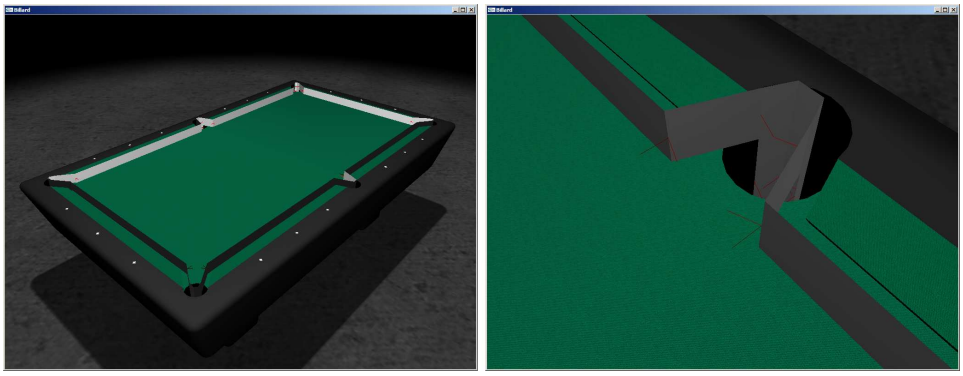

Abbildung 18: Darstellung der Kollisionsgeometrie, die um den Kugelradius r von der Bande entfernt modelliert wurde. In rot sind die Normalen dargestellt.

Da ein Kollisionstest mit dem kompletten Tisch, also insgesamt über 6000 Polygonen, viel zu aufwändig wäre, wurde er stark vereinfacht. Es wird nicht die tatsächliche Tischgeometrie getestet, sondern eine gesonderte Kollisionsgeometrie modelliert, sodass die Anzahl der Schnitttests auf 72 reduziert werden konnte. Um den Test, ob die Kugel die Bande bereits durchdrungen hat, noch weiter zu vereinfachen, wurde die Kollisionsgeometrie jeweils um r von der eigentlichen Geometrie entfernt modelliert, wobei r der Kugelradius ist. Anstatt den Abstand des Kugelmittelpunktes zur Bande zu messen genügt ein Test, ob sich der Kugelmittelpunkt hinter oder vor der Kollisionsgeometrie befindet. Abbildung 18 zeigt die Darstellung der Kollisionsgeometrie.

Die Kollisionsbehandlung mit den Banden entspricht einem recht einfachen Prinzip, das im Folgenden näher erläutert wird. Hierbei wird die Laufrichtung der Kugel als Strahl interpretiert, der mit jedem Dreieck der Kollisionsgeometrie geschnitten wird. Der vorderste Schnittpunkt wird dabei als Kollisionspunkt behandelt. Hat die Kugel die Bande durchschritten, wird die Kollision behandelt.

Der Methode *calculateRailCollision* werden dafür die Kugel, die Kollisionsgeometrie sowie der Queue für den Effet übergeben. Zunächst wird der vorderste Schnittpunkt der Kugel mit der Kollisionsgeometrie berechnet. Um diesen zu berechnen, wurde eine modifizierte Version des "Fast, Minimum Storage Ray-Triangle Intersection" Algorithmus (Quelle: [9]) verwendet. Grundlage dieses Algorithmus ist die Tatsache, dass jeder Punkt auf einem Dreieck mit Formel 20 berechnet werden kann.

$$\vec{x} = \vec{p}_1 + u \cdot (\vec{p}_2 - \vec{p}_1) + r \cdot (\vec{p}_3 - \vec{p}_1) \tag{20}$$

Dabei muss $u + r \leq 1$ gelten.

Zunächst werden zwei Seiten des Dreiecks, $\vec{e}_1$ und $\vec{e}_2$ bestimmt. Nun wird

zuerst getestet, ob die Kugellaufbahn parallel zum Dreieck verläuft. Hierzu wird ein neuer Vektor $\vec{p}$ berechnet mit $\vec{p} = \vec{v} \times e_1$. Jetzt kann über das Skalarprodukt zwischen $\vec{e}_1$ und $\vec{p}$ die Variable *det* berechnet werden. Ist $det = 0$ so läuft die Kugel parallel zum Dreieck, ansonsten wird weiter getestet. Dazu wird nun der Geschwindigkeitsvektor $\vec{v}$ mit Hilfe des Skalarprodukts auf eine der Edges projeziert, das Ergebnis ist der Parameter u. Ist $u < 0$ oder $u > 1$, so geht der Strahl auf jeden Fall am Dreieck vorbei, ansonsten wird weitergetestet. Nun wird $\vec{v}$ auf die andere Edge projeziert, das Ergebnis ist der Parameter r. Ist $v < 0$ oder $u + r > 1$, so geht der Strahl am Dreieck vorbei, andernfalls wurde ein Schnittpunkt gefunden. Abbildung 19 und Listing 2 zeigen den kompletten Vorgang.

Im nächsten Schritt wird geprüft, ob dieser Schnittpunkt der vorderste Schnittpunkt ist. Falls ja, wird die ID des Dreiecks und der Schnittpunkt selbst gespeichert. Dies ist nötig, da unter gewissen Umständen mehrere Schnittpunkte gefunden und die Berechnung dadurch verfälscht werden würde, wie in Abbildung 20 zu sehen ist.

Ist der vorderste Schnittpunkt bekannt, wird getestet, ob sich die Kugel vor oder hinter dem Dreieck befindet. Dies wird über das Skalarprodukt zwischen der Normale des Dreiecks und einem Vektor vom Dreieck zur Kugel ermittelt. Ist dieser Wert größer als 0, befindet sich die Kugel noch vor dem Dreieck, ansonsten hat sie das Dreieck bereits durchdrungen. Jetzt kann die eigentliche Behandlung der Kollision stattfinden.

Die einfachste Möglichkeit ist, den neuen Geschwindigkeitsvektor der Kugel nach dem Reflexionsgesetz mit Formel 21 zu berechnen (Quelle: [3]).

$$\vec{v} = \vec{v}_{old} - 2 \cdot (\vec{v}_{old} \circ \vec{n}) \cdot \vec{n} \tag{21}$$

Mit Hilfe dieser Methode ist es nicht möglich, den Effet der weißen Kugel zu behandeln. Aus diesen Grund wurde für diesen Sonderfall ein anderer Ansatz erdacht, der eine Modifizierung des Ausfallswinkels möglich macht. Zuerst wird der Betrag der Geschwindigkeit $|\vec{v}_{old}|$ ermittelt und für die spätere Berechnung gespeichert. Daraufhin wird mit Formel 22 der Winkel berechnet, in dem die Kugel auf die Bande prallt.

$$\alpha = \arccos\left(\frac{-\vec{v}_{old} \circ \vec{n}}{|\vec{v}_{old}| \cdot |\vec{n}|}\right) \cdot \frac{180}{\pi} \tag{22}$$

Nun muss unterschieden werden, ob die Kugel von links oder von rechts auf die Bande trifft, damit der Effet in die richtige Richtung aufaddiert wird. Hierzu dient ein Test mit dem Kreuzprodukt, wie in Formel 23 erläutert.

$$\vec{x} = \vec{v} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{23}$$

Ist die Y-Komponenten des Vektors $\vec{x}$ größer als 0, so trifft die Kugel von rechts auf die Bande, ansonsten von links. Um auch hier den Effet von der

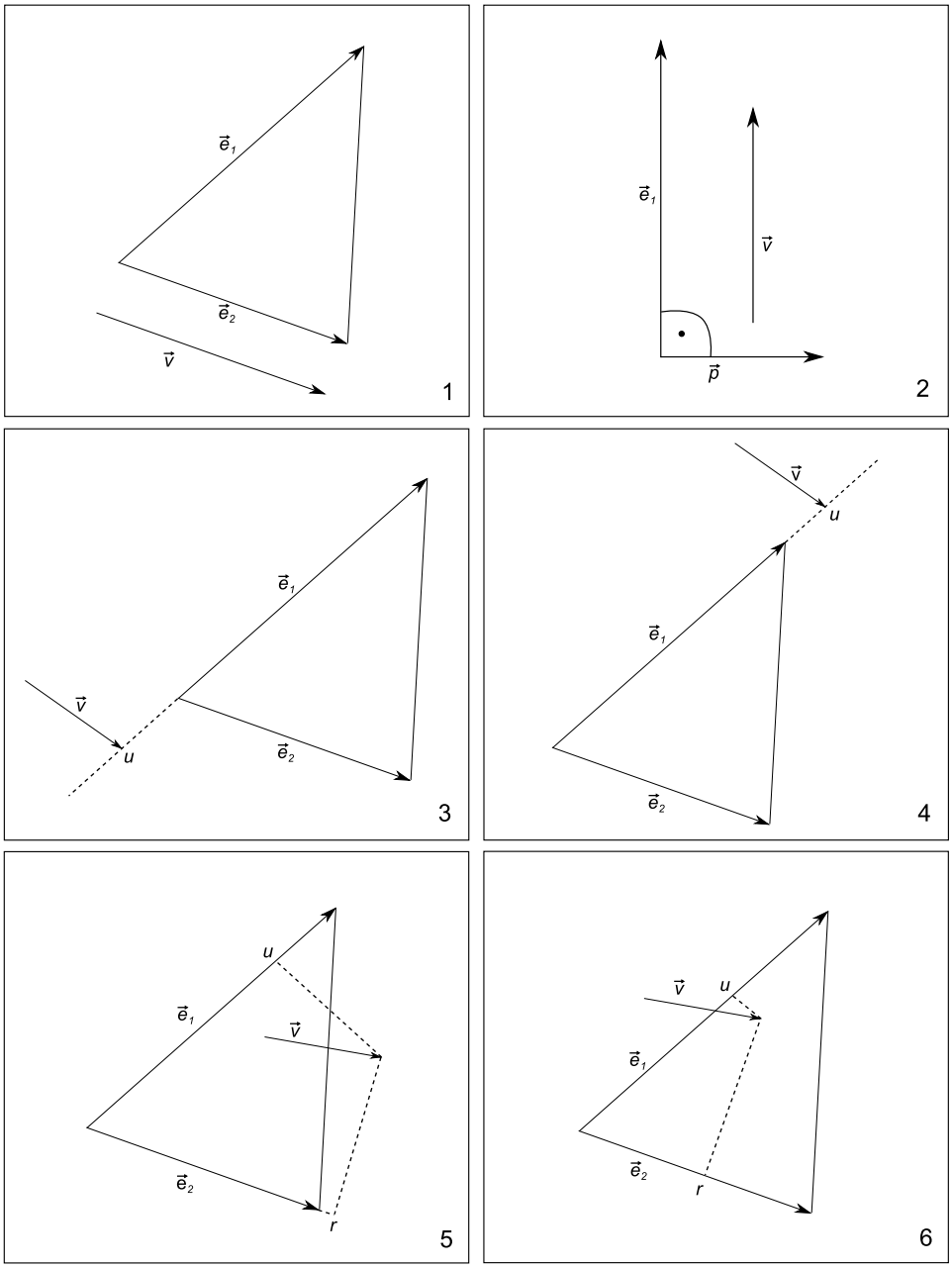

Abbildung 19: Schnitttest mit der Kollisionsgeometrie. In Bild 1 und 2 verläuft die Kugel parallel zum Dreieck, über den Hilsvektor $\vec{p}$ wird die Parallelität erkannt und der Schnitttest beendet. In Bild 3 ist $u < 0$, der Strahl geht also vorbei. In Bild 4 ist $u > 1$, der Strahl geht ebenfalls vorbei. In Bild 5 ist $0 < u < 1$, aber $u + v > 1$, der Strahl geht vorbei. In Bild 6 ist $u + v < 1$, der Strahl trifft das Dreieck.

Listing 2: Schnitttest zwischen Kugellaufbahn und Kollisionsgeometrie

```
Vec3D edge1, edge2, tVector, pVector, qVector;
float t, u, r, det, inv_det;
edge1 = trianglePoint2 - trianglePoint1;
edge2 = trianglePoint3 - trianglePoint1;
pVector = cross(velocity, edge2);
det = dot(edge1, pVector);

if(det > 0 || det < 0)
{
   inv_det = 1.0 / det;
   tVector = ballPosition - trianglePoint1;
   u = dot(tVector, pVector) * inv_det;
   if (u >= 0 && u <= 1)
   {
      qVector = cross(tVector, edge1);
      r = dot(velocity, qVector) * inv_det;
      if (r >= 0 && u + r <= 1)
      {
         t = dot(edge2, qVector) * inv_det;
         tempIntersectionPoint =
            ballPosition + t * velocity;
         float tempIntersectionDistance =
            length(tempIntersectionPoint - ballPosition);
         if (tempIntersectionDistance < intersectionDistance)
         {
            intersectionDistance = tempIntersectionDistance;
            intersectionTriangleId = triangleId;
            intersectionPoint = tempIntersectionPoint;
            intersectionFound = true;
         }
      }
   }
}
```

Schussstärke abhängig zu machen, wird wie bei der Kollisionsbehandlung zwischen zwei Kugeln der Geschwindigkeitsfaktor f errechnet und auf den Maximalwert 100 beschränkt. Nun kann der Ausfallswinkel β berechnet werden. Hierzu wird der Versatz des Queues in X-Richtung, e_x, auf den Winkel α addiert bzw. abgezogen, je nachdem, ob die Kugel von rechts oder links kam. Um keinen unnatürlichen Effet entstehen zu lassen, wird der Versatz des Queues vorher, wie in Formel 24 zu sehen, mit $f \cdot 0.3$ multipliziert, was sich in zahlreichen Tests als der optimale Skalierungsfaktor für ein gutes Spielgefühl erwiesen hat.

$$\beta = \alpha + e_x \cdot f \cdot 0.3 \tag{24}$$

β bleibt hier auf einen Maximalwert von 89 beschränkt. Der Effet wird natürlich mit jeder Kollision kleiner, daher wird er auch hier mit dem Faktor

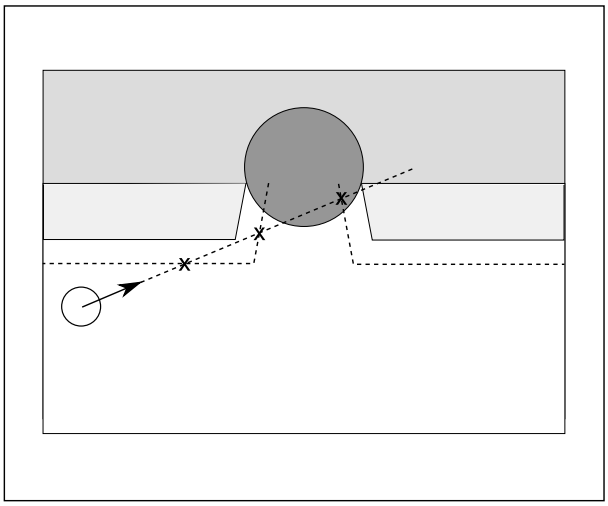

Abbildung 20: Besonders an den Ecktaschen können bei der Kollisionserkennung zwischen Kugel und Bande mehrere Schnittpunkte mit den Banden gefunden werden.

0.2 skaliert.

Um die neue Geschwindigkeit zu berechnen ein Hilfsvektor $\vec{h}$ benötigt, der sich wie folgt berechnet. Kommt die Kugel von rechts, wird Formel 25 verwendet, ansonsten Formel 26.

$$\vec{h} = \vec{n} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{25}$$

$$\vec{h} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \times \vec{n} \tag{26}$$

Jetzt kann mithilfe des Winkels β die Länge l dieses Vektors $\vec{h}$ ausgerechnet werden, die er haben muss um durch Addition der Normalen die neue Richtung zu ergeben. Dieser Vorgang ist in den Formeln 27 und 28 dargestellt.

$$l = \tan\left(\frac{\beta \cdot \pi}{180}\right) \tag{27}$$

$$\vec{v}_{new} = \vec{n} + \frac{\vec{h}}{|\vec{h}|} \cdot l \tag{28}$$

Abbildung 21 zeigt den bisherigen Vorgang noch einmal in der Übersicht. Da der Ausfallsvektor allerdings jetzt die falsche Länge hat, die Kugel also

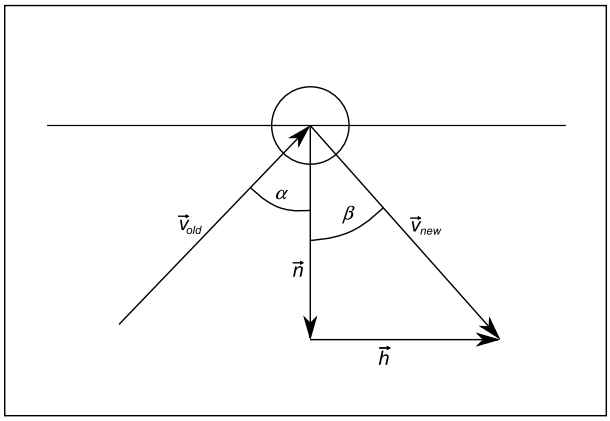

Abbildung 21: Behandlung einer Kollision mit der Bande.

entweder zu schnell oder zu langsam abprallen würde, wird der Vektor normalisiert und mit der Geschwindigkeit vor der Kollision multipliziert, wie in Formel 29 gezeigt.

$$\vec{v} = \frac{\vec{v}_{new}}{|\vec{v}_{new}|} \cdot |\vec{v}_{old}| \cdot 0.9 \tag{29}$$

Ist die Variable *isBouncing* dieses Bandenstücks auf *false* gesetzt, wird die Geschwindigkeit in X und Z-Richtung mit dem Faktor 0.01 multipliziert. Obwohl der Effet in Y-Richtung bei der Bandenkollision keine Auswirkungen hat, wird dieser noch invertiert und verringert, da durch die Kollision trotzdem ein Teil des Effets verloren geht.

Im letzten Schritt wird die Position der Kugel noch auf den gefundenen Schnittpunkt zurückgesetzt wie in Abbildung 22 gezeigt, da die Kugel sich sonst in ungünstigen Fällen auch bei der nächsten Berechnung noch hinter der Bande befinden kann und somit dort stecken bleibt.

Listing 3 zeigt die Implementierung dieses Algorithmus, Abbildung 23 zeigt die Kollisionsbehandlung im Spiel.

4.3.5 Einlochen

Es gibt mehrere Möglichkeiten, das Einlochen einer Kugel zu realisieren. Die physikalisch korrekte Variante wäre, die Spielfläche ebenfalls als Kollisionsgeometrie zu modellieren und eine Schwerkraft auf die Kugel wirken zu lassen. Dann würde ein Kollisionstest mit der Spielfläche durchgeführt, sodass die Kugel nur dort herunterfallen kann, wo auch das Spielfeld zuende bzw. eine Tasche ist. Da das Spiel aber in seiner jetzigen Form nur im zweidimensionalen Raum, also in der XZ-Ebene des Spielfelds, abläuft, wäre diese Variante eine unnötige Vergrößerung des Aufwands beim Schnitttest und bei der Berechnung der Physik.

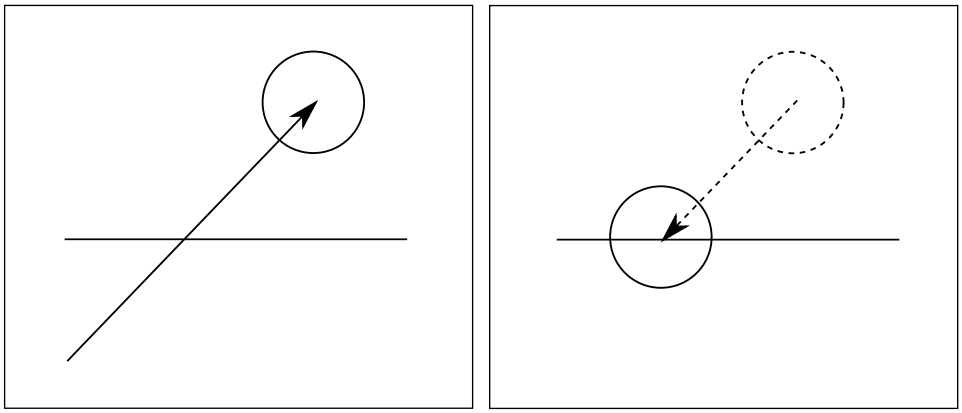

Abbildung 22: Zurücksetzen der Kugel, wenn sie die Bande bereits durchstoßen hat.

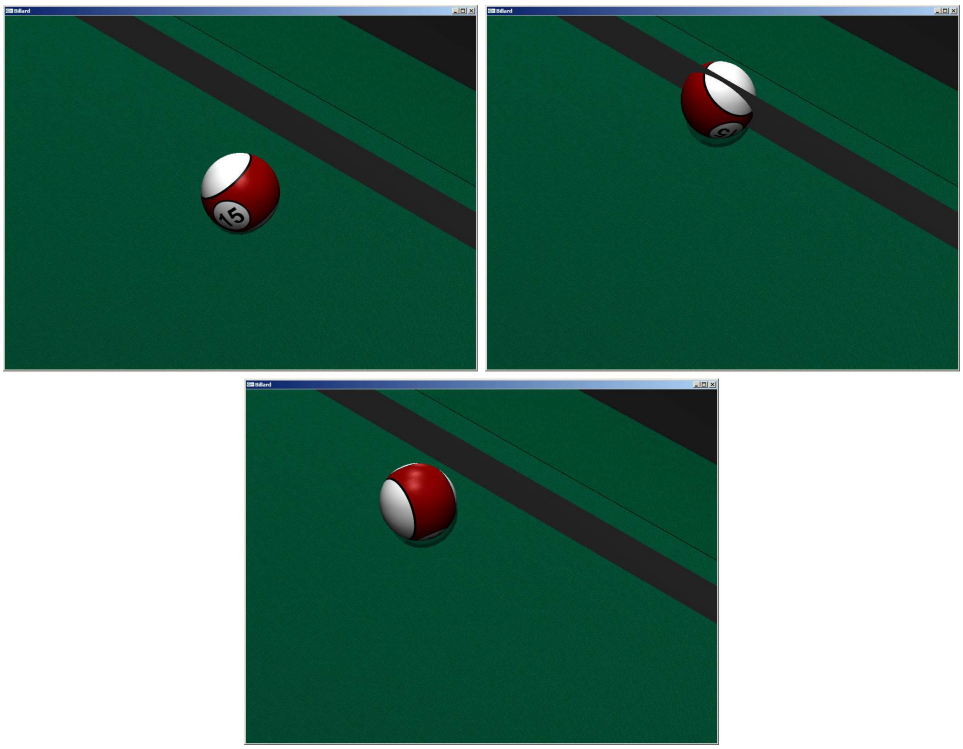

Abbildung 23: Kollisionsbehandlung im Spiel. Die Kugel läuft auf die Bande zu, die Kollision wird erkannt und die Kugel prallt von der Bande ab. Die Kollisionsgeometrie ist auf den Bildern visualisiert, um die Kollision mit ihr darzustellen.

Listing 3: Kollisionsbehandlung der Kollision mit einer Bande

```
1  Vec3D newDirection;
2  float velocitySize = length(ball->getLinearVelocity());
3  float angle = acos(dot(ballVelocity * -1, railNormal) /
4     (length(ballVelocity) * length(railNormal))) * 180.0f / PI;
5  Vec3D vec1;
6  Vec3D vCrossN = cross(ballVelocity, railNormal);
7  float velocityFactor = length(ballVelocity) / 4;
8  if (velocityFactor > 100) velocityFactor = 100;
9  if (vCrossN[Y] > 0)
10 {
11    if (ballNumber == 0)
12       angle -= queueDisplacementX * 0.3 * velocityFactor;
13    vec1 = cross(railNormal, Vec3D(0, 1, 0));
14 }
15 else
16 {
17    if (ballNumber == 0)
18       angle += queueDisplacementX() * 0.3 * velocityFactor;
19    vec1 = cross(Vec3D(0, 1, 0), railNormal);
20 }
21 if (angle > 89) angle = 89;
22 if (ballNumber() == 0)
23    queueDisplacementX *= 0.2;
24 float lengthVec1 = tan(angle * PI / 180.0) ;
25 vec1 = normalize(vec1);
26 vec1 *= lengthVec1;
27 newDirection = railNormal + vec1;
28 newDirection = normalize(newDirection);
29 if (!railIsBouncing)
30 {
31    newDirection[X] *= 0.01;
32    newDirection[Z] *= 0.01;
33 }
34 newDirection *= velocitySize * 0.9;
35 if (ballNumber() == 0)
36 {
37    queueDisplacementY *= -0.2;
38 }
39 ballVelocity = newDirection;
40 ballPosition = intersectionPoint;
```

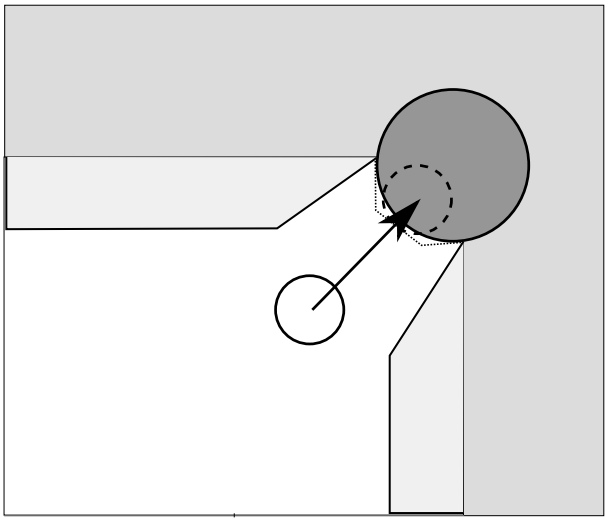

Abbildung 24: Kugel durchdringt eine Fläche und gilt dann als eingelocht.

Eine andere Variante wäre, weitere Geometrieobjekte zu modellieren, die eine Barriere darstellen, welche eine Kugel durchlaufen müsste, um als versenkt angesehen zu werden, wie in Abbildung 24 gezeigt. Dies würde allerdings wiederum einen größeren Aufwand bei der Kollisionsberechnung bedeuten, weshalb auch diese Variante in dieser Arbeit keine Anwendung findet.

Die dritte Variante, die für dieses Spiel am besten geeignet ist, beschränkt sich auf eine recht einfache Berechnung. Die Positionen sowie der Durchmesser der Taschen sind bekannt, somit kann ein Vektor von der Position der Kugel zum Taschenmittelpunkt, $\vec{d}$, bestimmt werden. Ist die Länge dieses Vektors kleiner als der Radius der Taschen, ist die Kugel eingelocht. Abbildung 25 verdeutlicht das Verfahren.

Ist $\vec{t}$ die Position der Tasche und $\vec{p}$ die Position der Kugel, lässt sich $\vec{d}$ berechnen über Formel 30.

$$\vec{d} = \vec{t} - \vec{p} \tag{30}$$

Der Durchmesser der Taschen beträgt 11, um ein besseres optisches Ergebnis zu erzielen wurde allerdings der Radius für den Test auf 4,5 reduziert. Ansonsten würde die Kugel, da nur ihr Mittelpunkt getestet wird, unter Umständen zum Teil durch den Tisch hindurch fallen.

In der Praxis kann dieses Verfahren noch weiter vereinfacht werden. Diese Berechnung muss nicht für jede der sechs Taschen durchgeführt werden, da der Tisch symmetrisch um den Ursprung herum aufgebaut ist und sich die beiden Mitteltaschen sowie die vier Ecktaschen nur in den Vorzeichen der Koordinaten unterscheiden. Wird also der Betrag der Kugelkoordinaten verwendet, reduziert sich die Anzahl der Berechnungen von sechs auf zwei, eine für die Mitteltaschen und eine für die Ecktaschen. Um die Leistung noch

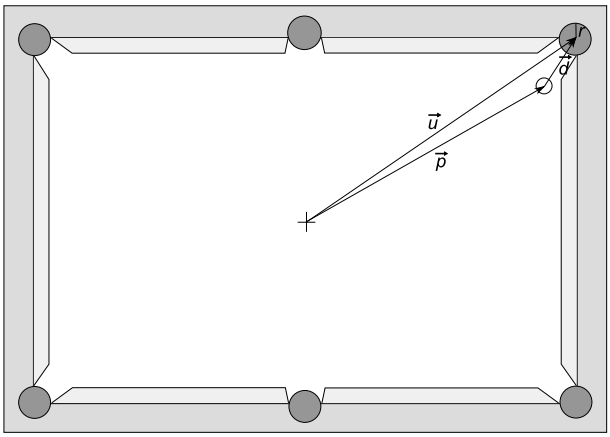

Abbildung 25: Der Vektor $\vec{d}$ vom Kugelmittelpunkt zum Taschenmittelpunkt wird ermittelt. Ist $|\vec{d}| \leq r$ gilt die Kugel als eingelocht.

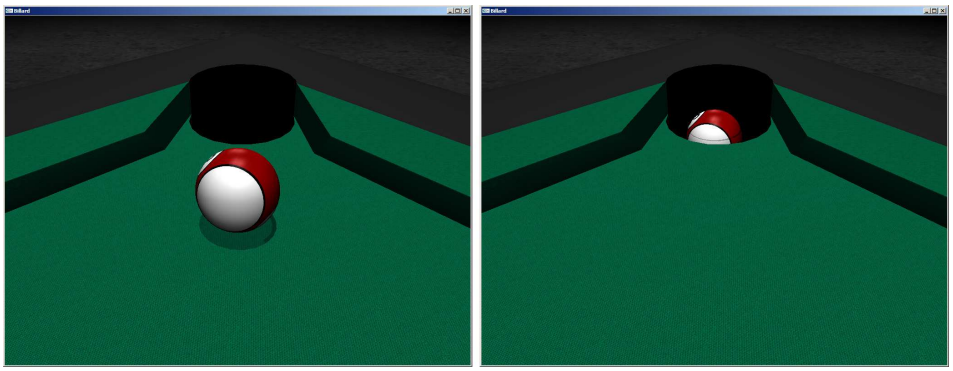

Abbildung 26: Vorgang des Einlochens im Spiel. Die Kugel läuft in eine Tasche und fällt nach unten.

etwas zu verbessern, wird dieser Test nur dann durchgeführt, wenn sich eine Kugel am Rand des Spielfelds befindet.

Wird eine Kugel als eingelocht betrachtet, wird ihr eine Geschwindigkeit $9.81 \cdot m$ entsprechend der Schwerkraft in negativer Y-Richtung zugewiesen. Fällt die Kugel unter den Schwellwert $y = -20$, wird ihre Sichtbarkeit auf *false* gesetzt und die Kugel somit nicht mehr gezeichnet. Listing 4 veranschaulicht die Implementation genau. Abbildung 26 zeigt den Vorgang während des Spiels.

4.4 Steuerung

Die Steuerung des Spiels sollte möglichst intuitiv gestaltet werden. Neben einer Rotation um die weiße Kugel sowie einem Zoom sollte es möglich sein, die weiße Kugel anzustoßen, den Anspielpunkt auf der weißen Kugel zu ver-

Listing 4: Test für das Einlochen der Kugeln

```
1 float absolutePositionX = fabs(ballPosition[X]);
2 float absolutePositionZ = fabs(ballPosition[Z]);
3 if (absolutePositionX > 117 || absolutePositionZ > 58)
4 {
5    if (length(Vec3D(124.5, 0, 65.5) -
6       Vec3D(absolutePositionX, 0, absolutePositionZ)) < 4.5)
7    {
8       velocity[Y] = -981 * mass;
9    }
10   else if (length(Vec3D(0, 0, 69) -
11      Vec3D(absolutePositionX, 0, absolutePositionZ)) < 4.5)
12   {
13      velocity[Y] = -981 * mass;
14   }
15 }
```

ändern, die Neigung des Queues festzulegen und die weiße Kugel nach einem Foul zu verschieben. Die eigentliche Steuerung sollte dabei komplett mit der Maus möglich sein.

Die einzelnen Möglichkeiten werden über verschiedene Modi realisiert. Die Idee dahinter ist einfach. Der Spieler befindet sich so lange im Beobachtungsmodus, bis er diesen verlässt um eine andere Aktion durchzuführen. Im Beobachtungsmodus kann er durch Bewegen der Maus bei gedrückter Maustaste die Kamera rotieren bzw. bei gedrückter rechter Maustaste zoomen. Drückt er nun beispielsweise die Taste "s", so verlässt er den Beobachtungsmodus und wechselt in den Schussmodus. Nun kann er bei gedrückter Maustaste den Queue vor und zurück bewegen, um letztendlich die Kugel anzustoßen. Diesen Modus kann er jederzeit wieder verlassen. Spätestens nach dem Anstoß wird automatisch in den Beobachtungsmodus gewechselt.

Folgende Modi stehen zur Verfügung:

- v: View-Modus. Die Kamera lässt sich um den *Center of interest* rotieren. Durch Drücken der rechten Maustaste und Bewegen der Maus wird gezoomt.
- s: Anstoß-Modus. Der Queue wird vor und zurück bewegt und so die weiße Kugel angestoßen.
- e: Anspielpunkt festlegen. Der Anspielpunkt des Queues auf der weißen Kugel wird verändert.
- m: Weiße Kugel bewegen. Die weiße Kugel kann auf dem Spielfeld verschoben werden.

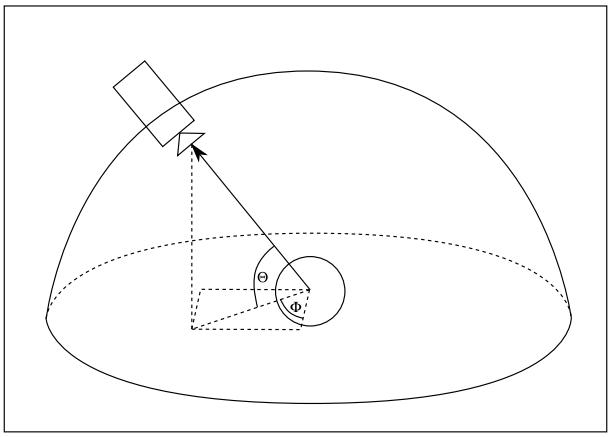

Abbildung 27: Rotation der Kamera um die weiße Kugel mittels Polarkoordinaten.

4.4.1 Rotation der Kamera

Die einfachste Methode zur Rotation der Kamera ist sicherlich das Verändern zweier Winkel und die folgende Rotation der Kamera um zwei Achsen mittels *glRotatef*. Da allerdings die Kameraposition für andere Teile des Programms, wie beispielsweise die Bestimmung der Stoßrichtung und die Platzierung des Queues, benötigt wird, fand diese Variante keine Anwendung. Stattdessen wurde die Rotation mittels *Polarkoordinaten* realisiert. Hierbei wird die Kameraposition direkt verändert, sodass sich die Kamera auf einer Halbkugel um den *Center of interest* drehen kann. Dafür werden zwei Winkel, Θ und Φ, benötigt, die die vertikale und horizontale Rotation der Kamera darstellen. Aus diesen Winkeln wird nach jeder Änderung die Kameraposition neu berechnet. Wichtig ist dabei, dass die beiden Winkel gewisse Grenzwerte nicht über- oder unterschreiten. Der Winkel Θ beispielsweise ist für die vertikale Rotation zuständig, darf also nur Werte zwischen 0° und 90° annehmen. Bei der Implementierung entspricht dies Werten zwischen 0 und $\frac{\pi}{2}$. Der Winkel Φ ist für die horizontale Rotation zuständig, was einem Wertebereich zwischen 0° und 360° entspricht, bei der Implementierung also 0 bis 2π. Listing 5 und Abbildung 27 verdeutlichen den Algorithmus.

4.4.2 Zoom

Für den Zoom wird zunächst ein Skalierungsfaktor bestimmt, der sich aus der Mausbewegung in Y-Richtung errechnet. Anschließend wird der Vektor vom *Center of interest* zur Kamera bestimmt und mit dem Skalierungsfaktor multipliziert. Zum Schluss wird noch getestet, ob die Kamera bereits eine definierte Entfernung über- oder unterschritten hat. Ist dies der Fall, wird die Kameraposition entsprechend korrigiert. So ist sichergestellt, dass der

Listing 5: Implementierung der Rotation der Kamera

```
1 theta -= 0.01 * deltaY * mouseSensitivity;
2 if (theta < 0.01) theta = 0.01;
3 else if (theta > (PI / 2) - 0.01) theta = (PI / 2) - 0.01;
4 phi += 0.01 * deltaX * mouseSensitivity;
5 if (phi < 0) phi += 2*PI;
6 else if (phi > 2*PI) phi -= 2*PI;
7 cameraPosX =
8     coi.getX() + length(cameraPos - coi) * sin(theta) * cos(phi);
9 cameraPosY =
10    coi.getY() + length(cameraPos - coi) * cos(theta);
11 cameraPosZ =
12    coi.getZ() + length(cameraPos - coi) * sin(theta) * sin(phi);
13 cameraPos.setXYZ(cameraPosX, cameraPosY, cameraPosZ);
```

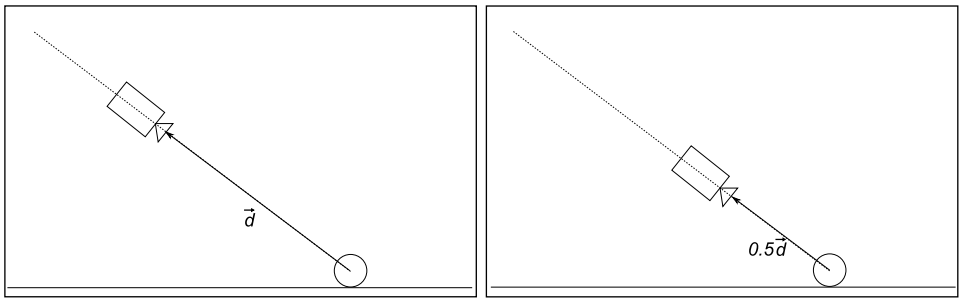

Abbildung 28: Die Kamera wird auf einer Geraden, die durch Kugel und Kamera verläuft, verschoben.

Spieler nicht zu nah oder zu weit von der Kugel weg zoomen kann. Listing 6 und Abbildung 28 verdeutlichen die Vorgehensweise.

4.4.3 Anspielen der weißen Kugel

Das Anspielen der weißen Kugel gestaltete sich als schwierigste Umsetzung der Steuerung. Der Queue kann hier mit der Maus vor und zurück bewegt werden, bis er die weiße Kugel berührt. Um zu bestimmen, wann der Queue die Kugel berührt, wird zunächst die Y-Koordinate der Maus gespeichert, sobald der Queue bewegt wird. Da der Queue durch Bewegen der Maus nach vorne und hinten gesteuert wird, ist nur die Y-Koordinate bei dieser Steuerungsart wichtig. Die gespeicherte Y-Koordinate stellt den Ausgangszustand des Queues dar, wird die Maus also weiter nach vorne bewegt als es bei Beginn der Bewegung der Fall war, berührt der Queue die weiße Kugel.

Wenn der Queue einmal nach vorne zur Kugel hin bewegt wird, wird die Entfernung zwischen der aktuellen Y-Koordinate der Maus und des Startwerts gemessen, sowie eine Zeitmessung gestartet. Wird der Queue zwischenzeitlich wieder nach hinten bewegt, werden diese Werte gelöscht und bei der

Listing 6: Implementierung des Kamerazooms

```
1 float scale = (100 - ((deltaY) * mouseSensitivity)) / 100;
2 cameraPos = coi + (cameraPos - coi) * scale;
3 if (length(cameraPos - coi) > 400)
4 {
5    cameraPos = coi + 400 * normalize(cameraPos - coi);
6 }
7 else if (length(cameraPos - coi) < 10)
8 {
9    cameraPos = coi + 10 * normalize(cameraPos - coi);
10 }
```

nächsten Vorwärtsbewegung erneut erfasst. Unterschreitet die Y-Koordinate der Maus den gespeicherten Startwert, wird die weiße Kugel angestoßen und ihr somit eine Geschwindigkeit zugewiesen. Diese Geschwindigkeit errechnet sich aus der Ausholweite des Queues d und der Zeit t, die dafür benötigt wurde, wie in Formel 31 dargestellt. Abschließend wird dieser Wert mit einer Konstanten multipliziert, damit die Geschwindigkeit der weißen Kugel ungefähr der Geschwindigkeit des Queues beim Anstoßen entspricht. Diese Konstante wurde in mehreren Testläufen ermittelt, um das bestmögliche Gefühl für den Queue zu übermitteln. Natürlich wäre es unrealistisch, wenn diese Geschwindigkeit ins Unendliche gehen könnte, da der Schussstärke auch in der Realität Grenzen gesetzt sind. Deshalb wird die Schussstärke auf 800 begrenzt.

$$s = \frac{d}{t} * 250 \tag{31}$$

Sobald die Schussstärke ermittelt ist, wird die Schussrichtung aus der Position der weißen Kugel und der Kamera errechnet, normalisiert und mit der Schussstärke multipliziert. Die so errechnete Geschwindigkeit wird dann der weißen Kugel zugewiesen und der Anstoß damit beendet. Sämtliche Aktionen werden für den Spieler gesperrt, bis die Kugeln wieder zur Ruhe gekommen sind, auch der Queue wird ausgeblendet. Lediglich der normale Beobachtungsmodus steht zur Verfügung, es ist also möglich, die Kamera um den vorherigen Mittelpunkt der weißen Kugel zu drehen bzw. zu zoomen. Abbildung 29 zeigt den Anstoßvorgang im fertigen Spiel.

4.4.4 Anspielpunkt festlegen

Der Anspielpunkt auf der weißen Kugel wird über zwei Variablen des Queues gesteuert, wie in Abbildung 30 gezeigt. Über diese Variablen wird der Queue später an der richtigen Position angezeigt und der Effet berechnet. Folglich reicht es natürlich aus, beim Festlegen des Anspielpunkts nur diese beiden Variablen entsprechend der Mausbewegung zu verändern. Eine Beschränkung sorgt dafür, dass der Queue nicht an der weißen Kugel vorbei stoßen

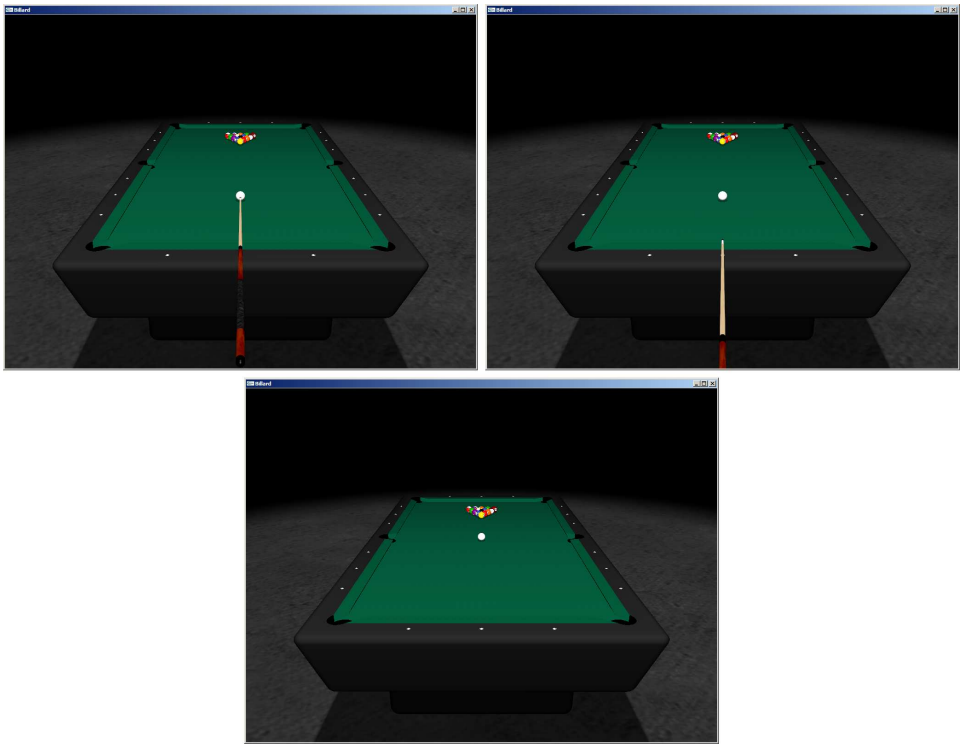

Abbildung 29: Anspielen der weißen Kugel. Links oben die Ausgangsposition, rechts oben wird mit dem Queue ausgeholt. Unten wurde die weiße Kugel angestoßen und der Queue ausgeblendet.

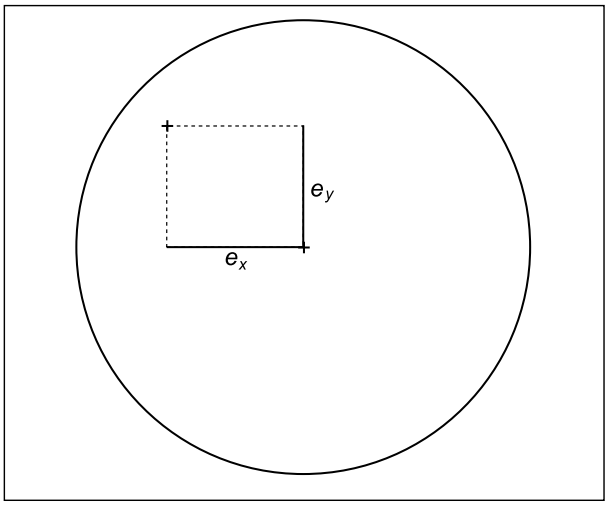

Abbildung 30: Über die Variablen e_x und e_y wird der Anspielpunkt auf der weißen Kugel bestimmt.

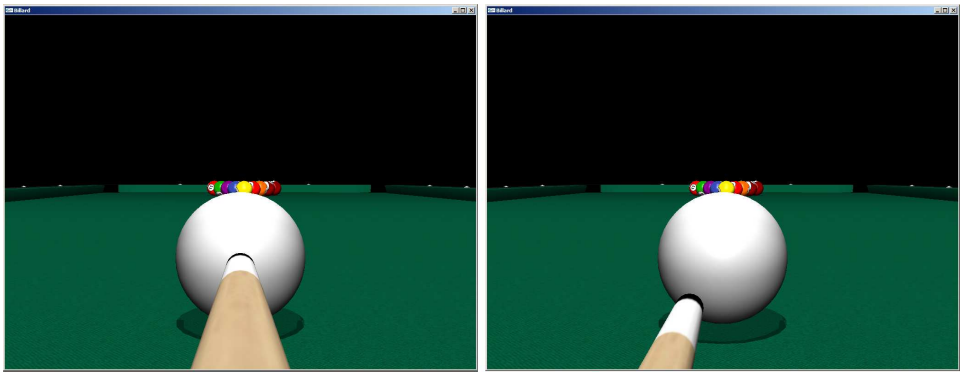

Abbildung 31: Festlegen des Anspielpunkts auf der weißen Kugel. Links die Ausgangsposition, rechts wurde der Anspielpunkt nach unten links verschoben.

kann. Wenn e_x der Versatz des Queues zum Mittelpunkt der weißen Kugel in X-Richtung, e_y der Versatz in Y-Richtung und $(\Delta x, \Delta y)$ die Mausbewegung ist, wird der neue Versatz mit Formel 32 und 33 berechnet. Abbildung 31 zeigt den Vorgang im fertigen Spiel.

$$e_x = e_{x_{old}} + (\Delta x) \cdot 0.01 \tag{32}$$

$$e_y = e_{y_{old}} - (\Delta y) \cdot 0.01 \tag{33}$$

4.4.5 Weiße Kugel verschieben

Zunächst wird der Vektor von der Kameraposition zum Mittelpunkt der weißen Kugel bestimmt, der Y-Wert dieses Vektors auf 0 gesetzt und anschließend normalisiert. Dieser Vektor gibt die Richtung nach vorne an ($\vec{f}$). Zu diesem Vektor wird ein senkrechter Vektor in der XZ-Ebene bestimmt ($\vec{s}$). Nun wird noch der Vektor vom Kugelmittelpunkt zur Kamera bestimmt ($\vec{c}$), um gleichzeitig auch die Kamera zu verschieben. Um die Kugel nun zu verschieben, werden diese beiden Vektoren mit der Mausbewegung $(\Delta x, \Delta y)$ multipliziert und auf die Position der weißen Kugel addiert, wie auch in Abbildung 32 und Formel 34 gezeigt:

$$\vec{p} = \vec{p}_{old} - \Delta y \cdot \vec{f} + \Delta x \cdot \vec{s} \tag{34}$$

Eine Beschränkung in den X und Z-Koordinaten sorgt dafür, dass die Kugel dabei nicht über die Spielfläche hinaus bewegt werden kann. Die neue Kameraposition wird schließlich über die neue Kugelposition sowie den Vektor $\vec{c}$ mit Formel 35 berechnet.

$$\vec{p}_{cam} = \vec{p} + \vec{c} \tag{35}$$

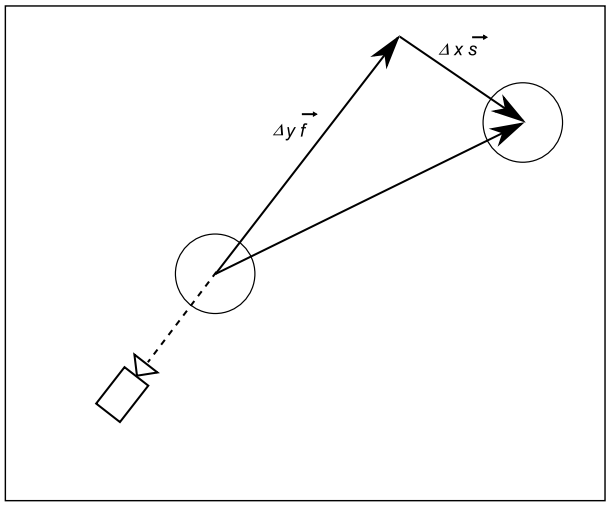

Abbildung 32: Verschieben der weißen Kugel mit Hilfe zweier Vektoren $\vec{f}$ und $\vec{s}$.

4.5 Darstellung

Zur Beleuchtung der Szene wurden vier Lichtquellen verwendet. Drei hängen in einer Höhe von 90cm über dem Tisch, die vierte befindet sich zentral in einer Höhe von einem Zentimeter über der Spielfläche. Die Farbwerte sowie Helligkeiten der Lichtquellen wurden so gewählt, dass die Szene nicht übermäßig hell oder dunkel erscheint.

Die Darstellung des Tisches, der Kugeln und des Queues wird in den folgenden Kapiteln erläutert.

4.5.1 Tisch

Nachdem der Tisch in *Maya* modelliert war, wurde er mittels eines *Maya File Loaders* von Mirko Geissler (Quelle: [7]) ins Programm integriert. Der *File Loader* besteht aus zwei Komponenten, dem *Maya Exporter* und dem *OpenGL Importer*. Der *Maya Exporter* liegt als Plugin für *Maya* vor und ermöglicht es, Szenen in einem eigenen Dateiformat abzuspeichern. Dabei werden alle den Tisch betreffenden Informationen direkt mit abgespeichert, wie die Geometrie, die Materialien und die Texturen. Der *OpenGL Importer* schließlich lädt das Objekt aus der Datei und zeigt es mittels einer eigenen *draw* Methode an. Abbildung 33 zeigt den Tisch im Programm.

4.5.2 Kugeln

Die Kugeln werden über eine eigene draw Methode dargestellt. Dabei wird die Kugel zunächst an ihre Position verschoben und in die richtige Orientierung gedreht. Die Orientierung wird hierzu vom Quaternion in eine Axis

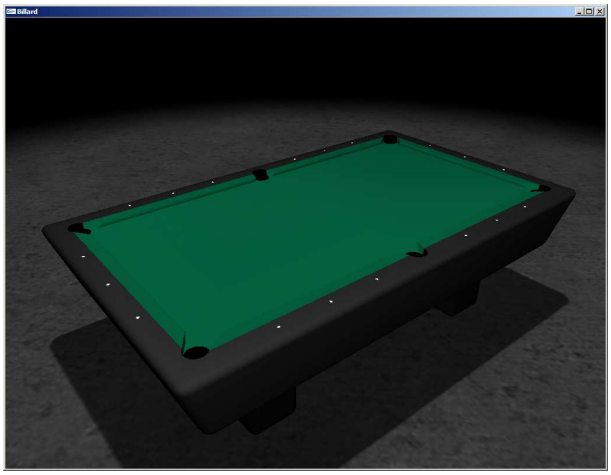

Abbildung 33: Der fertig geladene Tisch im Spiel.

Listing 7: Umrechnung von der Orientierung als Quaternion in eine Axis Angle Rotation

```
1  float rotationAngle = 2 * acos(orientation[W]);
2  Vec3D rotationAxis;
3  if (rotationAngle != 0)
4  {
5      rotationAxis = orientation[Z] / sin(rotationAngle / 2);
6  }
7  else
8  {
9      rotationAxis = Vec3D(1, 0, 0);
10 }
11 glRotatef(180 * rotationAngle / PI, rotationAxis[X],
12     rotationAxis[Y], rotationAxis[Z]);
```

Angle Rotation umgerechnet, analog zur Erstellung des Quaternions. Der Winkel berechnet sich aus Formel 36, die Rotationsachse aus Formel 37.

$$\alpha = 2 \cdot \arccos(q_w) \tag{36}$$

$$\vec{a} = \frac{q_{\vec{z}}}{\sin(\frac{\alpha}{2})} \tag{37}$$

Listing 7 zeigt den genauen Implementationscode. Anschließend wird die Kugel als *Quadric Object* mit der passenden Textur erzeugt. Um eine runde Form und gute Beleuchtungseffekte zu erzielen, besteht jede Kugel aus 40x40 Polygonen und besitzt einen Radius von 2.86. Listing 8 zeigt den genauen Code. Zum Schluss wird noch der Schatten gerendert. Der Schatten besteht aus einer einfachen in Y-Richtung auf 0.01 skalierten schwarzen Kugel mit *Alpha-Blending*. Diese Methode ist zwar sehr einfach, aber auch sehr effektiv. Abbildung 34 zeigt die Kugeln im fertigen Spiel.

Listing 8: Code zur Erzeugung einer Kugel als Quadric Object

```
1 GLUquadricObj *qobjB;
2 qobjB = gluNewQuadric();
3 gluQuadricDrawStyle(qobjB, GLU_FILL);
4 gluQuadricNormals(qobjB, GLU_SMOOTH);
5 gluQuadricTexture(qobjB, true);
6 glBindTexture(GL_TEXTURE_2D, mId);
7 gluSphere(qobjB, 2.86, 40, 40);
```

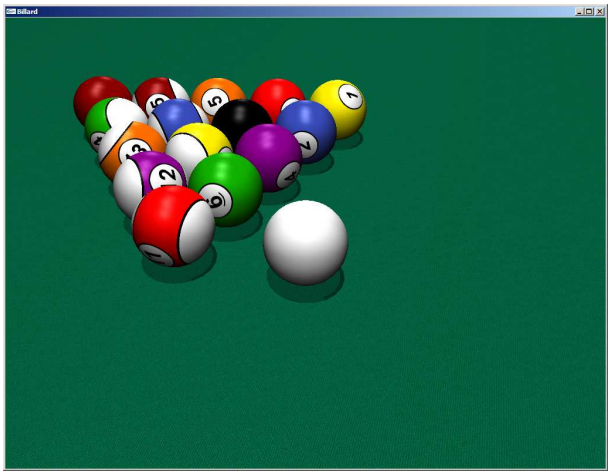

Abbildung 34: Darstellung der Kugeln im Spiel.

4.5.3 Queue

Der Queue besteht aus zwei *Quadric Objects*, einem Zylinder und einer Kugel. Zunächst wird der Queue zur Kamera hin gedreht. Dazu wird der Rotationswinkel um die Y-Achse benötigt, der wie folgt berechnet wird. Zunächst werden zwei Hilfsvektoren $\vec{h}_1$ und $\vec{h}_2$ benötigt, die sich aus der Kameraposition $\vec{p}$ und dem *Center of interest* $\vec{c}$ mit Formel 38 und 39 berechnen lassen. Anschließend kann der Rotationswinkel mit Formel 40 berechnet werden.

$$\vec{h}_1 = \begin{pmatrix} p_{cam_x} - c_x \\ 0 \\ p_{cam_z} - c_z \end{pmatrix} \tag{38}$$

$$\vec{h}_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{39}$$

$$\alpha = \frac{180 \cdot \arccos(\vec{h}_1 \circ \vec{h}_2)}{|\vec{h}_1| \cdot |\vec{h}_2| \cdot \pi} \tag{40}$$

Wenn $p_{cam_x} > c_x$ ist, wird der Winkel negiert, da der Queue ansonsten entgegen der Kamerarotation rotiert wird. Nun wird der Queue im Winkel α

Listing 9: Code zur Darstellung des Queues

```
1 Vec3D vec1 = cameraPos - coi;
2 vec1[Y] = 0;
3 Vec3D vec2 = Vec3D(0, 0, 1);
4 float alpha = (180 / PI) * acos(dot(vec1, vec2) /
5    (length(vec1) * length(vec2)));
6 if (cameraPos[X] > coi[X]) alpha *= -1;
7 glTranslatef(whiteball[X], whiteball[Y], whiteball[Z]);
8 glRotatef(-alpha, 0, 1, 0);
9 glRotatef(mGrade, 1, 0, 0);
10 glTranslatef(mDisplacementX, mDisplacementY, mPosition);
11
12 glPushMatrix();
13 glTranslatef(0, 0, 145);
14 GLUquadricObj *qobjB;
15 qobjB = gluNewQuadric();
16 gluQuadricDrawStyle(qobjB, GLU_FILL);
17 gluQuadricNormals(qobjB, GLU_SMOOTH);
18 gluQuadricTexture(qobjB, true);
19 mTexture2->useTexture();
20 gluSphere(qobjB, 1.26, 20, 20);
21 glPopMatrix();
22
23 GLUquadricObj *qobj;
24 qobj = gluNewQuadric();
25 gluQuadricDrawStyle(qobj, GLU_FILL);
26 gluQuadricNormals(qobj, GLU_SMOOTH);
27 gluQuadricTexture(qobj, true);
28 mTexture1->useTexture();
29 gluCylinder(qobj, 0.5, 1.25, 145, 20, 1);
30 glPopMatrix();
```

um die Y-Achse und in der Neigung um die X-Achse gedreht und zur weißen Kugel hin verschoben. Die Neigung des Queues ist rein optischer Natur, sie hat noch keine Auswirkungen auf den Anstoß. Da der Queue jetzt noch in der weißen Kugel stecken würde, wird er entlang seiner Orientierung um den Faktor d von der weißen Kugel weg verschoben. Beim Anspielen der weißen Kugel wird später nur dieser Wert geändert, um ein Anstoßen zu simulieren. Der Zylinder, aus dem der Queue besteht, ist an beiden Enden offen, er muss also zumindest an dem Ende, das zur Kamera zeigt, mit einer Kugel verschlossen werden. Diese Kugel wird genau so rotiert und verschoben wie der Zylinder, wird aber am Ende statt um den Faktor d um den Faktor $145+d$ verschoben. Der Zylinder besitzt eine Länge von 145, der Durchmesser am Stoßende beträgt 0.5, der Durchmesser am Griff 1.25. Die Kugel hat einen leicht größeren Durchmesser von 1.26, um sonst teilweise auftretende Lücken zu schließen. Listing 9 veranschaulicht die Implementierung der Darstellung des Queues.

5 Fazit

Zusammenfassend lässt sich sagen, dass fast alle Ziele an diese Studienarbeit erfüllt wurden. In sechsmonatiger Entwicklungszeit entstand ein Billardspiel mit einer möglichst realistischen Physikengine, die Möglichkeiten zur Kollisionserkennung zwischen allen Objekten beinhaltet. Ebenso wurde eine Ballphysik implementiert, die das Rollen der Kugeln darstellen, sowie die Berechnung der Kraftverteilung bei einer Kollision durchführen kann. Es ist zudem möglich, die weiße Kugel mit Effet anzuspielen, sodass sich die Laufrichtung der weißen Kugel nach einer Kollision ändert. Eine intuitive Steuerung konnte ebenso implementiert werden. Es werden nahezu alle Aktionen mit der Maus durchgeführt, die Tastatur dient lediglich dem Wechsel zwischen den einzelnen Modi. Es ist möglich, die weiße Kugel mit der Maus anzuspielen, den Anspielpunkt auf der weißen Kugel festzulegen sowie diese zu verschieben. Des weiteren kann die Kamera um die weiße Kugel gedreht werden, ein Zoom ist ebenfalls implementiert. Ferner kann die Neigung des Queues verändert werden, was allerdings bisher nur optische Auswirkungen hat. Der Tisch ist ebenfalls möglichst maßstabsgetreu modelliert worden, wobei kurz vor Fertigstellung dieser Arbeit aufgefallen ist, dass der Tisch 9cm zu kurz modelliert wurde. Die Texturierung aller Objekte ist ebenfalls passend.

Passende Soundeffekte wurden nicht implementiert, da es mangels Aufnahmegeräten sowie Billardequipments nicht möglich war, eigene Soundeffekte aufzunehmen. Ebenso wurden die Spielregeln wegen Zeitmangel nicht mehr implementiert. Die Unterstützung der Stereoleinwand wurde aus demselben Grund nicht mehr implementiert.

6 Ausblick

Wie nahezu jedes Programm könnte auch dieses Spiel ständig erweitert und verbessert werden. Hier bieten sich natürlich zunächst die nicht erreichten Ziele an, die Integrierung von Soundeffekten sowie Spielregeln. Eine Anzeige, welche Kugeln sich für den jeweiligen Spieler noch auf dem Tisch befinden, wäre ebenfalls wünschenswert. Darüber hinaus wäre die Modellierung neuer Tische in verschiedenen Größen sowie eine bessere Texturierung möglich. Auch die Umgebungen, in denen die Tische stehen, können ausmodelliert werden. Ebenso wäre es möglich, das Spiel durch die Verwendung einiger Shader deutlich schöner aussehen zu lassen. Richtige Glanzlichter auf den Kugeln sowie *Bump-Mapping* für den Tisch wären hier denkbar. Eine Erweiterung der Physikberechnungen auf den dreidimensionalen Raum würde auch Sinn machen, sodass die Kugeln nicht nur rollen, sondern auch springen können. Damit könnten auch *Massestöße* simuliert werden, die in der jetzigen Physikengine nicht zu berechnen sind. Ebenso könnten weitere Spielarten

wie beispielsweise *Snooker* oder *14.1 endlos* implementiert werden. Ein eigenständiges Thema für eine Studienarbeit wäre zudem die Implementierung einer künstlichen Intelligenz für einen virtuellen Gegenspieler in verschiedenen Schwierigkeitsstufen.

Abbildungsverzeichnis

Literatur

[1] PhD David G. Alciatore. The illustrated principles of pool and billiards. http://www.engr.colostate.edu/~dga/pool/.

[2] Mark DeLoura. *Game Programming Gems*. Charles River Media, 2000.

[3] Ralf Engels. Reflexion, Sep. 1997. http://theorie.informatik.uni-ulm.de/Lehre/WS9798/Computergrafik/Engels/node10.html.

[4] Deutsche Billard-Union e.V. Spielregeln Pool, Mar. 2003. http://www.billard-union.de/public/regelwerk/files/Regeln_Pool_Turnier.pdf.

[5] Deutsche Billard-Union e.V. Materialnormen Pool, Dec. 2004. http://www.billard-union.de/public/regelwerk/files/Materialnormen_Pool.pdf.

[6] Peter T. Früh. Objektorientierte Entwicklung einer Billard-Simulation. Technical report, Zürcher Hochschule Winterthur, Feb. 2001.

[7] Mirko Geissler. Entwicklung einer 3D Anwendung mit erweiterter optischer und haptischer Unterstützung. Technical report, Universität Koblenz-Landau, 2006 Sep.

[8] Jeff Lander. Physics on the back of a cocktail napkin. *Game Developer*, Sep. 1999.

[9] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[10] Rick Parent. *Computer Animation - Algorithms and Techniques*. Morgan Kaufmann, Jan. 2002.

[11] Wipke Schulte. Fachliche und didaktische Ausarbeitung der Vorlesung Mechanik und Wärmelehre, Sep. 1999. http://www.physik.rwth-aachen.de/group/IIIphys/INFOS/Exscript/4Kapitel/IV6Kapitel.html.

[12] Shreiner, Woo, Neider, and Davis. *OpenGL Programming Guide*. Addison Wesley, 4 edition, Feb. 2004.

[13] Sobeit Void. Quaternion powers, Feb. 2003. http://www.gamedev.net/reference/articles/article1095.asp.